Multitech

MPF-III BASIC Programming Manual

Micro-Professor III BASIC Programming Manual

COPYRIGHT

DISCLAIMER

MULTITECH INDUSTRIAL CORP.
Office: 315 Fu Hsin N. Rd., Taipei, Taiwan, R.O.C.
Factory: 1 Industrial E. Rd., III Hsinchu Science-based Industrial Park, Hsinchu, Taiwan, R.O.C.

PREFACE

This manual is written based on the assumption that the user has some fundamental knowledge of **BASIC** (Beginners All-purpose Symbolic Instruction Code). The contents of this manual provides the user with an extended version of MPF-III **BASIC** or what is simply known as **MBASIC**.

Although it may not be absolutely necessary for the user to have had an introduction to BASIC in order to read this manual, it is recommended that the user first read the **THE INTRODUCTION TO MICRO-PROFESSOR III BASIC PROGRAMMING** which is the introductory manual that precedes this version of extended MBASIC. Since instructions on how to install MPF-III are contained in the **INSTALLATION MANUAL** and description of MPF-III's peripheral equipment/devices are discussed in the MPF-III **USER'S MANUAL**, this manual is primarily concerned with explaining how to use MPF-III to write and run MBASIC programs.

The first chapter of this manual provides a brief overview of some of the syntactic rules and definitions as well as some fundamental program commands used in MBASIC. This chapter gives a quick review of some of MBASIC's rules to those who have already read the introductory manual to MBASIC.

In Chapters 2 through 8, commands related to particular programming topics are discussed along with some examples to help you understand how the commands function. You may want to try some of the examples and the sample programs on your MPF-III to get a feel for how the commands can be used to do what you want. For your quick reference and for the sake of convenience, the commands have been alphabetized in each chapter. A complete alphabetized list of the summary of MBASIC commands is provided in Appendix A.

Chapter 9 discusses some useful math functions which you may need to use in some of your programs. In this chapter, a list of which built-in functions and derived functions are available in MBASIC has been itemized in an easy-to-read fashion for your convenience.

Many useful appendices are provided for improving your programming efficiency. Listings of error messages and codes are also included in the appendices to help you locate errors and correct them. The contents of the appendices are especially inserted just to help you understand how MBASIC functions on MPF-III.

Overall, this manual could prove to be a useful tool for the user who has previously had a brief introduction to MBASIC. The set-up of this manual's text and appendices have been so devised to make quick-referencing an easier task.

Now let's get started...

HAVE FUN PROGRAMMING IN MBASIC!

CONTENTS

APPENDICES

CHAPTER 1
INTRODUCTION TO MPF-III BASIC

In this first chapter, our discussion on the introduction to the MPF-III BASIC will generally cover how to get MBASIC started up, the basic rules in MBASIC programming, and some fundamental program commands. Although you may want to skip over this chapter since it seems to be a brief review of what was covered in the Introduction to Micro-Professor III BASIC Programming Manual, it is recommended that you read this chapter to pick up on the extra details and syntactic definitions which have been added to certain programming concepts. You will find that this chapter is useful in providing you with the basic groundwork and the means to get "warmed up" in Micro-Professor III BASIC programming.

1.1 INITIATION AND EXECUTION OF MPF-III BASIC

To become familiar with MPF-III BASIC, you are suggested to have the MPF-III installed and running while reading this manual. We recommend that you follow the procedures described in the **USER'S MANUAL** to set up the MPF-III.

After the MPF-III is hooked up, you will see a prompt character "]" followed by a cursor (which is a white square against a black background) on the TV or monitor screen. The prompt indicates where an input line can be typed, while the cursor points to the position where a character can be inputted.

When the screen displays "]", your MPF-III is running in BASIC mode, and you can type in BASIC programs. The BASIC language for use with the MPF-III is called MPF-III BASIC or MBASIC.

Note:

1) The presence of the prompt character represents two things: (a) It tells you that you can type in lines of statements or commands, and (b) it tells the user the language in which the MPF-III is using. When the MPF-III runs in MBASIC, the prompt character "]" is displayed. When the MPF-III is in the monitor mode, the prompt character "*" is displayed.

2) By typing CALL -159 and [←┘], the monitor prompt character "*" will be displayed on the screen. To return to MBASIC from the monitor you can type CONTROL C and [←┘], and the MBASIC prompt "]" will appear.

1.2 SYNTACTIC RULES

In this section, symbols or expressions used to represent certain concepts in MBASIC programming are discussed. Since these expressions are used throughout this manual, it is recommended that you read this section carefully. A summary of the most commonly used metasymbols are listed in Appendix B.

1.2.1 Syntactic Definitions and Abbreviations

The definitions listed below use some metasymbols such as |, [,], {, }, \. These metasymbols are not intrinsic to MBASIC. They are simply used to express the structure or relationships in MBASIC. A special symbol ":=" indicates the beginning of a definition of the term which is to the left of the := mark.

```
|      := Alternatives are separated with this metasymbol
[ ]    := Items enclosed in this metasymbol are optional
{ }    := Items enclosed in { } can be repeated
\      := Items whose values are to be used are enclosed
          in \  \.  The value of x is written \x\
~      := The metasymbol that indicates a required space

metasymbol
       := ||[|]|{|}|\|~

lower-case letter
       := a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|x|
          y|z|

metasymbol
       := lower-case letter

digit
       := 1|2|3|4|5|6|7|8|9|Ø

metaname
       := {metasymbol}[digit]

metasymbol
       := a single digit adjoined to a metaname

special
       := !|#|$|%|&|'|(|)|*|:|=|-|@|+|;|?|/|>|.|<|,|]|^|"
          Control characters (characters which are typed
```

while holding down the CONTROL key) and the null character are also special symbols.

letter
:= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

character
:= letter|digit|special

alphanumeric character
:= letter|digit

name
:= letter [{letter|digit}]
A name may consist of up to 238 characters in length. MBASIC differentiates one name from another by the first two characters and ignores the characters that follow the first two. Thus, RMCHECK and RMTEST are the same name to MBASIC. Reserved words and the quote mark (") cannot be used in a name.

integer
:= [+|-]{digit}
Integers must be in the range -32767 to 32767. When converting non-integers into integers, MPF-III truncates the non-integer to the next smaller integer. But, when the value of a non-integer is very close to the next larger integer, the rule mentioned above is not applicable. For example,

SCREEN

```
A% = 651.999 999 959 999
PRINT A%
652

B% = 783.999 999 96
PRINT B%
784

C% = 32333.34
PRINT C%
32333

D% = 6789.999 996
PRINT D%
6789
```

integer variable name
:= name%
A real may be stored as an integer variable; however, MBASIC will first convert the real into an integer.

real
:= [+|-]{digit}[.{digit}][E[+|-]digit[digit]]
:= [+|-][{digit}].[{digit}][E[+|-]digit[digit]]
The letter E here means exponent or *10^. The value of the real is the result of the number to the left of E multiplying the number which is 10 raised to the power specified by the number to the right of E. Reals must be in the range from -1E38 to 1E38. If a real is not in this range, you will receive the message ?OVERFLOW ERROR. However, when performing the operation of addition or subtraction, you may get numbers as large as 1.7E38 without getting the overflow message.

If the absolute value of a real is less than 2.9388E-39, MBASIC considers the real to be a zero.

The following are considered by MBASIC to be reals when presented by themselves and are therefore evaluated as zero:

```
.    +.    -.    .E     +.E    -.E
.E+     .E-    +.E-    +.E+    -.E-
```

Thus, the array element M(.) is the same as M(0).

In addition to the abbreviated reals listed above, the following are recognized as reals and evaluated as zero when used as numeric responses to INPUT and as numeric elements of DATA.

```
+    -    E    +E    -E    SPACE
E+   E-   +E+  +E-   -E+   -E-
```

The GET instruction evaluates all of the single-character reals above as zero.

When printing a real, MBASIC prints at most nine digits. Any extra digits are rounded off.

Leading zeroes in front of the first non-zero digit before the decimal point are not printed. Trailing zeroes to the right of the last non-zero digit after the decimal point are not printed as well. If there is no zero after the decimal point, then the decimal point is not printed.

If the absolute value of a real is greater than or is equal to .01 and is less than 999 999 999.2, then the real is printed in fixed point notation. If the real is in the range .0 100 000 000 5 to .0 999 999 999, the real is printed with up to ten digits, including the zeroes immediately following the decimal point.

If you attempt to type in a number with more than 39 digits such as

```
SCREEN
123.12345678901234567890123456789012345678
```

You will receive the following message

```
SCREEN
?OVERFLOW ERROR
```

even though this number is in the range -1E38 to 1E38. The same will happen if you use the following number

123.0000000000000000000000000000000000000000

However, MBASIC ignores leading zeroes.
If the first digit is a one, and the second digit equals to or is less than six, numbers with 39 digits may still be used without getting the error message. It takes five bytes in memory to store a real number.

real variable name
    := name

arithmetic variable
:= avar

avar
:= name | name%
All simple varibles occupy 7 bytes in memory--two bytes for the name and five bytes for the real or integer value.

delimiter
:= ¯|(|)|=|-|+|^|<|>|/|*|,|;|:
A name and reserved word may not be separated by these delimiters.

arithmetic operator
:= aop

aop
:= +|-|*|\|^

arithmetic logical operator
:= alop

alop
:= AND|OR|=|>|<|<>|><|>=|=>|<=|=<
(Note that "NOT" has been purposely left out)

operator
:= op

op
:= aop|alop

arithmetic expression
:= aexpr

aexpr
:= avar|real|integer
:= (aexpr)
If the depth of nested parentheses exceeds 36, the MPF-III will show ?OUT OF MEMORY ERROR.
:= [+|-|NOT]aexpr

subscript
:= (aexpr[{,aexpr}])
The maximum number of dimensions is 89. aexpr must be positive and is converted to integer when used.

avar

:= avar subscript

aexpr

:= avar subscript

literal

:= [{character}]

string

:= "[{character}]"

A string occupies one byte for its length, two bytes for its location pointer, and one byte for each character in the string.

:= "[{character}]return

String of this kind is used only at the end of a line.

null string

:= " "

:= A string which contains no characters

string variable name

:= name$

string variable

:= svar

svar

:= name$|name$subscript

The location pointer and variable name each occupy two bytes in memory. The length and each string character occupy one byte.

string operator

:= sop

sop

:= +

string expression

:= sexpr

sexpr

:= svar|string

:= sexpr sop sexpr

string logical operator

:= slop

slop
:= =|>|>=|<|<=|=<|<>|><

aexpr
:= sexpr slop sexpr

variable
:= var

var
:= avar|svar

expression
:= expr

expr
:= aexpr|sexpr

prompt character
:=]
When the "]" symbol appears, MBASIC is ready to receive a command.

reset
:= A press of the RESET key on the keyboard.

return
:= A press of the carriage return ↵ key on the keyboard.

ctrl
:= hold down the CONTROL key while the following named key is pressed.

line number
:= linenum

linenum
:= {digit}
The line number must be in the range from Ø to 65279. Otherwise, the message ?SYNTAX ERROR occurs.

line
:= linenum [{instruction:}]instruction return
A line may consist of up to 239 characters, including spaces typed in by the user. However, spaces added internally by MBASIC are not included.

1.2.2 Rules for Evaluating Expressions

An expression combines constants, variables, or functions with operators in an ordered sequence. When evaluated, an expression must result in a value.

An operator performs a mathematical or logical operation on one of two values resulting in a single value. Generally, an operator is placed between two values. Operators tell a computer the type of operation to perform on values in an expression. Some examples of expressions are:

```
(A*3) - (B + 2)

(X*(Y - 2)) + Z
```

Listed below are the operators used in MBASIC:

Highest

```
( )
+ - NOT (unary operators)
^
* /
binary +, binary -
Relational > < >= <= =< ><=
AND
OR
```

Lowest

The operator at the highest level is performed first, followed by any other operators in the hierarchy shown above. If operators are at the same level, the order is from left to right. Operations enclosed in parentheses are performed before any operations outside the parentheses. When parentheses are nested, operations within the innermost pair are performed first.

Operators may be divided into two types depending on the kind of operation performed. The main types are arithmetic, relational, and logical (or Boolean) operators. There are unary operators that precede a single value. For example, the minus sign in A - B is a binary operator that results in subtraction of the two values A and B; the minus sign in -A is a unary operator indicating that A is to be negated.

1.2.3 Conversion of Number Variable Formats

When integers and reals are both used in an operation, all numbers are converted to reals before the calculation takes place. The results are converted to the arithmetic type (integer or real) of the final variable to which they are assigned. Functions which are defined on a given arithmetic type will convert arguments of another type to the type for which they are defined. Let's look at the following program:

```
SCREEN
10 A%=21
20 B=2.3
30 C%=A%+B
40 PRINT "C%="; C%
]RUN

C%=23
```

In the above program, A% and C% are integer variable names, and B is a real variable name. When MPF-III executes the instruction

```
C%=A%+B
```

in line 30, it will discover that A% is an integer and B is a real. Therefore, to perform the arithmetic operation, it will first consider the value of A% to be a real (even though it really isn't) and add it to the value of B, thereby determining the result to be 23.3. However, since the result is to be stored in the integer variable C%, then the final value of C% is determined to be 23.

1.2.4 Execution Modes

There are two execution modes in MBASIC--immediate execution mode and deferred execution mode.

(1) imm Immediate-execution mode. In MBASIC some instructions may be used for this mode. In this mode, an instruction is entered without a line number. When the RETURN key is pressed, the instruction is executed immediately.

Try the following examples.

Enter

SCREEN

```
PRINT 1Ø-4
```

Then press the RETURN key.
The MBASIC responds with

SCREEN

```
6
```

From the above example, note that the PRINT command is executed immediately after pressing the RETURN key.

Enter

SCREEN

```
PRINT 1/5, 3*4
```

Press RETURN.
The MBASIC responds with

SCREEN

```
.2            12
```

(2)def Deferred-execution mode. Instructions used in this mode must be typed in a line beginning with a line number. When the RETURN key is pressed, MBASIC stores the numbered lines for later use. Instructions in this mode are executed only when a RUN command is entered following the last line of a program. The line number should be in the range from Ø to 65279.

Try the following.

Enter

SCREEN

```
10 PRINT 5+6
20 PRINT 5-6
```

(Don't forget to press the RETURN key to terminate an input line.)

A program consisting of a series of such instructions as the one above is entered in deferred execution mode. When a program is entered, the MPF-III stores the program in its memory, but does not execute the program immediately. MBASIC executes a program starting from the lowest line number through the highest line number immediately after you type RUN and RETURN.

Now type

SCREEN

```
RUN
```

MBASIC prints

SCREEN

```
11
-1
```

In the example above, we first enter the line with the line number 10, and then enter the line with the line number 20. However, it makes no difference in what order you type deferred-execution statements. MBASIC puts the statements in correct numerical order.

In order to examine the program and the order of statements currently stored in memory, you may type

SCREEN
```
LIST
```

And MBASIC responds with

SCREEN
```
10 PRINT 5+6
20 PRINT 5-6
```

To delete a line of a program, type the line number of the line you intend to delete and then press the RETURN key.

Type

SCREEN
```
10 [←]

LIST
```

The MBASIC responds with

SCREEN
```
20 PRINT 5-6
```

You have deleted line 10 from your program. To insert a new line 10, type 10 and the new statement which is supposed to be executed. Now type

SCREEN
```
10 PRINT 5-5
LIST
```

The MBASIC responds with

```
SCREEN

    10 PRINT 5-5
    20 PRINT 5-6
```

To change or correct a program line, you do not have to delete a line first and then insert a new program. The easiest way to correct a program line is to type the new line number (in this example 10), followed by the new statements. The MBASIC will automatically replace the old line with the new one.

It is a good habit to leave blank lines in a program. Blank lines enable you (the programmer) to insert additional program lines whenever you think it is necessary. An increment of 10 between line numbers is generally sufficient.

To erase the program currently in memory, type the NEW command. If you no longer want to use the program currently in memory and intend to write a new program, do not forget to enter the NEW command. Entering this command prevents the intermingling of the old program with the new program.

Now type

```
SCREEN

    NEW
```

MBASIC responds with the prompt character

```
SCREEN

    ]
```

Then type

```
SCREEN

    LIST
```

MBASIC responds with

```
SCREEN
    ]
```

showing that the old program has been deleted.

1.2.5 Number Format

In MBASIC, reals must be in the range from -1*1Ø^38 to 1*1Ø^38. Reals beyond this range will cause the MPF-III to generate error messages. However, when performing addition or subtraction, you may sometimes generate numbers as large as 1.7*1Ø^38 without getting error messages. If the absolute value of a number is less than 3*1Ø^-39, MBASIC considers the number to be zero.

In addition to the above limitations, integers must be in the range from -32767 to 32767. When a number is to be printed, it is printed according to the following rules.

1) If the number is negative, the minus sign is printed.
2) If the absolute value of the integer is in the range from Ø to 999999999, it is printed as an integer.
3) If the absolute value of the number is equal to or greater than Ø.Ø1 and less than 999999999.2, it is printed in fixed point notation without the exponent.
4) If the number does not fall under categories 2 and 3, the number is printed in scientific notation.

If a number is printed in scientific notation, the format is as follows:

												S	x	x
S	dØ	.	d1	d2	d3	d4	d5	d6	d7	d8	E			

The leading S represents the sign of the number. If the number is positive, the positive sign is not printed. Otherwise, the minus sign is printed. Each digit in the above number format is an integer from Ø to 9. A non-zero digit is printed in front of the decimal point. Up to eight digits can be printed after the decimal point. E stands for exponent, followed by the sign (S) of the exponent and the exponent itself (xx). Leading zeroes and trailing zeroes are never printed,

that is, no zero is printed before or after the decimal point.

If the digits after the decimal point are all trailing zeroes, then the decimal point is not printed. The sign for the exponent is always printed. It may be either a "+" or "-". The exponent always appears in the form of two digits, i.e. zero is not omitted in the exponent field.

The value of the number that is printed in scientific notation is the result of the number to the left of E multiplying 10 raised to the power of the number to the right of E.

The following are examples of numbers and the printed format used by the MBASIC.

NUMBERS	OUTPUT FORMAT
+3	3
-5	-5
-78.650	-78.65
4321	4321
39.67E3	39670
1*10^10	1E+10
-9.5876431*10^20	-9.5876431E+20
10000000	1E+07
37645812	37645812

A number typed on the keyboard or a numeric constant used in MBASIC may have as many as 39 digits. The first ten digits are significant, and the 10th digit is rounded off. For instance, if you type

SCREEN

```
PRINT 12345678900987654321
```

MBASIC responds with

SCREEN

```
1234567890
```

1.2.6 Variable Names

A name, when used in a computer program, is usually used to identify a memory location in a computer.

MBASIC recognizes names beginning with an alphabetic character (from A through Z), followed by alphanumeric characters (which is any letter from A through Z, or any digit from 0 through 9.).

A variable name may consist of 238 characters. However, MBASIC distinguishes variable names from one another, using only the first two characters of a variable name. Therefore, MBASIC interprets CAT and CATMEOW to be identical names.

Reserved words in MBASIC are not allowed to be used as a variable name or part of a variable name. All the commands in MBASIC are reserved words. Thus, LDATA cannot be used as a name because DATA is a reserved word. All the reserved words in MBASIC are listed in Appendix F.

Compare legal and illegal variable names listed below:

LEGAL	ILLEGAL
PS	PFOR
TA	TO
CT	
N1%	

You may use the INPUT, LET commands or assignment statement to assign a value to a name. Try the following examples:

SCREEN

```
10 A = 3
20 PRINT A,A*A+2
]RUN
3         11
```

```
SCREEN

    10 LET A = 10
    20 PRINT A,A/5
    ]RUN
    10         2
```

As can be seen from the examples, LET is optional in an assignment statement. The BASIC language uses this statement to memorize the values of variables, that is, to store the values of variables in memory.

Under the following four conditions below, the values of variables can be erased to make spaces available for new incoming data (values).

1) A new line is entered into the program or an old line is deleted.
2) A CLEAR command is given.
3) A RUN command is issued.
4) The NEW command is entered.

All variable names for numbers are considered to be zero by MBASIC unless a number is assigned to it. For example:

```
SCREEN

    PRINT A+5, A*5, A
```

The MBASIC replies with

```
SCREEN

    5         0         0
```

In previous examples, the REM statement can be added. REM stands for REMark. Comments are usually added following REM so it is easier for a programmer to understand what is going on in a program. The comments or notes following REM are ignored by the MBASIC.

1.2.7 Real, Integer and String Variables

In MBASIC, variables come in three types--real, integer, and string variable. Reals are printed with up to nine digits of accuracy and may range up to 10 to the 38th power. When MBASIC performs an operation, it first converts a real from decimal to binary form for internal calculation. When the MBASIC is requested to print the results, it converts the binary back to the decimal. Because of errors caused by rounding off numbers and other unpredictables, MBASIC internal math subroutines; such as square root, division and exponentiation, do not always give the precise number, but will instead give an approximate number as the resulting answer.
The number of digits to the right of the decimal point may be set by rounding off the value prior to PRINTing it. The formula for achieving this is shown as below:

```
A = INT(A*10^N + .5)/INT(10^N + .5)
```

In the above formula, N stands for the number of decimal places, and A is the number to be printed.

A faster way to accomplish this is to modify the formula as follows:

```
A = INT(A*B + .5)/B
```

In this case, B = 10^N. Where B = 10 is one place, B = 100 is two places, and B = 1000 is three places, etc. to the right of the decimal point. The formula works for A >= 1 and A < 999999999.

The three types of variables used in MBASIC are summarized in the table below.

DESCRIPTION	SYMBOL TO APPEND TO VARIABLE NAME	EXAMPLE
Strings (Ø to 255 characters)	$	MPF$ A$
Integers (must be in range of -32767 to +32767)	%	X%
Real precision (Exponent -38 to +38, with 9 decimal digits)	none	TAX

An integer variable must be followed by the symbol "%", while a string variable is always ended with the symbol "$". A$, A%, and A are three different variables.

Integer variables cannot be used in FOR and DEF statements. However, if you are interested in saving storage space, it would be advantageous to use integer variables in array operations whenever possible.

All arithmetic operations are performed in reals. Before calculation takes place, all integers and integer variables were converted to reals. The functions SIN, COS, TAN, SQR, LOG, EXP, and RND also convert their arguments to reals and generate such results.

When a number is converted to an integer, it is truncated or rounded down.

For example:

```
X% = Ø.999        Y% = -Ø.Ø1
PRINT X%          PRINT Y%
Ø                 -1
```

If an integer variable is assigned a real number, PRINTing the integer variable has the same effect as if the INT function had been applied. However, no automatic conversion can be done between strings and numbers. Thus, assigning a number to a string variable results in an error message. Nevertheless, there are some special functions that can convert one type to the other.

1.2.8 Arrays

An array is a table consisting of numbers. The name of an array is the same as the name of the table. MBASIC is devised so that any single element of an array can be accessed.

In a program, the array name X is distinct and different from the simple variable name X. Therefore, the array name and variable name can be used in the same program.

SCREEN

```
10 I = 1
20 PRINT "2*";I;"=";2*I
30 I = I + 1
40 IF I <> 11 THEN 20
]RUN

2*1=2
2*2=4
2*3=6
2*4=8
2*5=10
2*6=12
2*7=14
2*8=16
2*9=18
2*10=20
```

In this program, the variable I is initialized with the value of 1 in line 10. In line 20, the PRINT instruction will cause MPF-III to print out the multiplication table of 2's, perform the multiplication and print the product. Line 30 will increment the current value of I by 1 each time line 20 is executed. In line 40, IF...THEN will cause MPF-III to loop back to line 20 until the value of I is 11. When I = 11, the execution of this program will stop.

Among computer programmers, this concept of repeatedly executing specific instructions is known as "looping" and is used extensively in programming in BASIC. Another way to perform the task of "looping" is to use the FOR ...NEXT instruction as shown below:

To distinguish various elements in an array, a subscript is used. The I'th element in the array A is represented by A(I). I is the subscript we used to distinguish one element from another. Note that A(I) is only one element in an array. In this manual, only one-dimensional arrays are discussed.

The exact dimension of an array can be set by the DIM A(X) statement, where DIM stands for DIMensions and X determines the space of the array. An array subscript always starts at 0. Thus, DIM A(12) allocates an array whose elements range from Ø through 12.

If the dimension of an array has not been set by the DIM A(X) statement, MBASIC automatically assumes that the dimensions of the array contain eleven elements with subscripts ranging from Ø to 1Ø.

The following program requests a user to input 1Ø numbers in succession and then print the 1Ø numbers.

```
1Ø DIM A(1Ø)
2Ø FOR N = 1 TO 1Ø
3Ø INPUT A(N)
4Ø NEXT N
5Ø FOR N = 1 TO 1Ø
6Ø PRINT A(N);" ";
7Ø NEXT N
```

When line 1Ø is executed, MBASIC allocates 11 positions for the A array--A(Ø) through A(1Ø). Line 2Ø through line 4Ø requests a user to input 1Ø elements which will be put into the A array. Line 5Ø through 7Ø prints the elements in the A array in accordance with the order they were put into the array. Note that a subscript may be any expression.

1.2.9 Strings

A sequence of characters is generally known as "literal". A literal enclosed in quotation marks is a string. The following examples are strings:

"MPF-III"
"CHRISTMAS"

Values can be assigned to string variables the same way they can be assigned to numeric variables. However, a string variable must be appended with the special symbol "$". Take a look at the following example below:

SCREEN

```
A$ = "HELLO CHRIS"
PRINT A$
HELLO CHRIS
```

In the above example, the string value "HELLO CHRIS" is assigned to the string variable A$. When the command (PRINT A$) is given, MPF-III will print the literal in string A$. Now that you know how to assign a string value to a string variable A$, there are times when you may need to know the length of a string variable (the number of the characters in a string). To find out how many characters are in a string, type the following:

SCREEN

```
PRINT LEN(A$), LEN("HELLO CHRIS")
12              11
```

The number of characters in a string may range from 0 to 255. A string without a character is a "null string". Before a string variable is assigned a string value, it is considered to be a null string by MBASIC. Printing a null string on the screen will cause nothing to be printed, and the cursor will not move to the next column. Try the following:

SCREEN

```
PRINT LEN(X$);X$;7
Ø7
```

Another method to create a null string is to use an assignment statement:

```
X$ = "" or
LET X$ = ""
```

Setting a string variable to an null string can free up the string space used by a non-string variable. But you will get into trouble by assigning a null string to a string variable. Detailed discussion of this problem can be found in Chapter 6 under the IF statement.

In most cases, only a portion of a string variable is used. For instance, suppose we only want to use the first five letters of the string A$. To extract the first five letters of string A$, type in the following command:

SCREEN

```
PRINT LEFT$(A$,5)
```

After execution of this command, MPF-III will then print

SCREEN

```
HELLO
```

which is the first five letters of the string, "HELLO CHRIS".

LEFT$(A$,N) is a string function which returns the leftmost N letters in the string A$.

For example:

SCREEN

```
10 A$="HELLO CHRIS"
20 FOR I = LEN(A$) TO 1 STEP-1
30 PRINT LEFT$(A$,N)    I
40 NEXT I
50 FOR J = 1 TO LEN(A$)
60 PRINT RIGHT$(A$,J)
70 NEXT J
]RUN

HELLO CHRIS
HELLO CHRI
HELLO CHR
HELLO CH
HELLO C
HELLO
HELLO
HELL
HEL
HE
H
S
IS
RIS
HRIS
CHRIS
 CHRIS
O CHRIS
LO CHRIS
LLO CHRIS
ELLO CHRIS
HELLO CHRIS
```

Because the string A$ has 11 characters, the first FOR...NEXT loop in the above example will be executed 11 times, where I=1,2,...11. The first time the loop is executed, all the characters will be printed. The second time, the ten leftmost characters (including space) will be printed and so on. Likewise, the second FOR...NEXT loop in the above program will also be executed 11 times, where J=1,2,...11.

Another string function, RIGHT$, can be used to return the rightmost characters from the string function A$. In the above example try using RIGHT$(A$,N) in place of LEFT$(A$,N) and observe what happens.

In addition to the two previously mentioned string functions, MID$ is another useful string function. The string function MID$ allows you to take characters from the middle of the string and print them out. The following example demonstrates how this string function works.

```
FOR K=1 TO LEN(A$):PRINT MID$(A$,K):NEXT K
```

By using MID$(A$,K), you will be able to return a substring beginning at the Kth position of A$ and ending at the last character of A$. In string functions, position 1 is the very first possible position and position 255 is the last possible position.

In any string function, it is possible that you only want to extract the Kth character. In this case, the MID$ string function can be used with three arguments as follows: MID$(A$,K,1). The third argument represents the number of characters to be returned starting with character K.

For example:

SCREEN

```
FOR K=1 TO LEN(A$):PRINT MID$(A$,K,1),MID$(A$,K,2)

NEXT K

        H    HE
        E    EL
        L    LL
        L    LO
        O    O
              C
        C    CH
        H    HR
        R    RI
        I    IS
        S    S
```

For more details on LEFT$, RIGHT$ and MID$, refer to Chapter 5 of this manual.

In addition to these functions, strings may also be concatenated (put or joined together) by using the plus(+) operator, as shown in the following example:

SCREEN

```
B$=A$ + "," + " " + "HOW ARE YOU?"
PRINT B$
HELLO CHRIS, HOW ARE YOU?
```

If you would like to take a string apart, modify it and then put it back together again, concatenation is the most useful and convenient method to accomplish this task. For example:

SCREEN

```
C$=MID$(B$,7,5)  + "-" + LEFT$(B$,5) +  "-"  +
     RIGHT$(B$,12)
PRINT C$
CHRIS-HELLO-HOW ARE YOU?
```

Sometimes you may wish to convert a digit into a string function and vice-versa. The performance of this task can be accomplished by using the functions VAL and STR$, which is illustrated in the example below:

SCREEN

```
STRING$="123.4"
PRINT VAL(STRING$)
123.4
```

SCREEN

```
STRING$=STR$(5.3637)
PRINT STRING$,LEFT$(STRING$,5)
5.3637        5.363
```

The STR$ function can be utilized to change numbers to a desired format for input or output. To convert a

number to a string, you can use LEFT$, RIGHT$, MID$ and concatenation to reform it into the desired format.

STR$ may also be used to find out how many print positions a number will use. For example:

SCREEN

```
PRINT LEN(STR$(55555.379))
9
```

Now try the example below by entering the following question:

WHAT IS THE AREA OF A RECTANGLE OF LENGTH 6.47 INCHES AND 6.2 INCHES?

In this example, the VAL function may be used to extract the numeric values 6.47 and 6.2 from the question. For more information on CHR$ and ASC, refer to Chapter 5 of this manual.

1.3 FUNDAMENTAL PROGRAM COMMANDS

In this section, program commands fundamental to MBASIC programming are discussed. Many of these commands are used extensively in programming and will appear in many of the programs contained in this manual. Some of the commands, such as the IF...THEN instruction or FOR...NEXT loop, are discussed in more detail in later chapters. For the most part, these fundamental program commands are presented in this chapter to get you started and to give you a preview of what to expect as you advance into other computer topics covered in this manual.

1.3.1 Print Format

In displaying information, it is sometimes desirable to format data or text into columns or separated by spaces. The use of a comma (,) in a PRINT statement will cause spacing over to the next tab field to occur. Therefore, the value following the comma will be printed after it has moved to the next tab field. If a semicolon (;) is used in place of a comma, the next value will be printed right next to the previous value. Try the following example:

SCREEN

```
PRINT 3,6,9
3    6    9

PRINT 3;6;9
369

PRINT -3;-6;-9
-3-6-9
```

The following example, as contrasted to the one above, is a sample program, which reads a value from the keyboard and uses the value for calculation. After the value of the formula has been calculated, the resulting product will be printed upon execution of the RUN command.

SCREEN

```
10  INPUT R,H
20  PRINT 3.14159*R*R*H

RUN
?10,5
1570.795
```

When MBASIC executes the INPUT instruction, it displays a question mark (?) on the screen to prompt for an input by the user from the keyboard (in this example, 10 and 5 were typed). The variable following the INPUT statement is assigned the typed value (in the example above, the INPUT value of R is set to 10 and H is set to 5). Program execution then continues with the next statement which is line 20 in the above program.

When the formula after the PRINT statement is evaluated, the variable R will be substituted by the value 10 each time R appears in the formula and H will be substituted by the value 5. Therefore, the formula becomes 3.14159*10*10*5.

If you would like to calculate the value of various data, we could keep re-running the program. However, an easier way to do this is by simply adding another line to the previous program as follows:

SCREEN

```
    GOTO 10

RUN

?10,5
1570.795

?3,7
197.92017

?47,50
346988.615

?
BREAK IN 10

]
```

By inserting a GOTO statement at the end of the above program, you have caused the program to loop back to line 10 after it prints each answer. This could go on forever, but the execution of the above program was interrupted after calculating three pairs of data in the above example. Stopping the input of data was achieved by typing a CONTROL C (typing C while holding down the CONTROL key) and pressing the RETURN ↵ key. Through the use of CONTROL C, you can stop the execution of any program after executing the current instruction.

1.3.2 IF...THEN

Suppose we would now like to write a program to test whether or not a typed number is less than zero. The simplest way to test whether or not a typed number is less than zero is to use the IF...THEN statement, which provides a conditional branch to another statement.

Type

SCREEN

```
NEW
```

to delete your last program and then type the following program:

SCREEN

```
10  INPUT A
20  IF A<0 THEN 50
30  PRINT "VALUE OF A IS";A
40  GOTO 60
50  PRINT "VALUE OF A IS ERROR"
60  END
```

When this program is executed, a question mark will be printed and the computer will wait for you to enter a value for variable A. Type any number at random. The computer will then execute the IF instruction. Notice that there is an "assertion" between the IF and the THEN portion of the statement. An assertion consists of

two expressions separated by one of the following symbols:

SYMBOL	MEANING
=	EQUAL TO
>	GREATER THAN
<	LESS THAN
<> OR ><	NOT EQUAL TO
<=	LESS THAN OR EQUAL TO
>=	GREATER THAN OR EQUAL TO

The IF statement is determined to be either true or false, depending on whether the assertion is true or not. In our current program above, if -1 is typed for A the assertion A<Ø is true. Thus, the IF statement is true, and program execution continues with the THEN portion of the statement, which is to go to line 5Ø. In line 5Ø, the instructions are to print VALUE OF A IS ERROR and the execution of line 6Ø will end the program.

Suppose we run this program again and type a value of 1 for A. The assertion A<Ø is now false (therefore, the IF statement in line 2Ø is false) and the execution of the program will continue with the next line number, ignoring the THEN porion of the statement and any other statements in that line. In line 5Ø, the VALUE OF A IS and the value of A will be printed and then the GOTO statement in line 4Ø will send the computer down to line 6Ø to end the program.

Now let's try the following program (remember: type NEW first, to delete your current program):

SCREEN

```
10  INPUT "INPUT ENGLISH SCORE";I
20  IF I > 85 THEN 50
30  IF I < 60 THEN 60
40  PRINT "*** LEVEL:FAIR ***"
45  PRINT : GOTO 10
50  PRINT "*** LEVEL:EXCELLENT ***"
55  PRINT : GOTO 10
60  PRINT "*** LEVEL:POOR ***"
65  PRINT : GOTO 10
]RUN
```

```
INPUT ENGLISH SCORE 9Ø
*** LEVEL:EXCELLENT ***

INPUT ENGLISH SCORE 75
*** LEVEL:FAIR ***

INPUT ENGLISH SCORE 5Ø
*** LEVEL:POOR ***

INPUT ENGLISH SCORE 67
*** LEVEL:FAIR ***
```

The above program has been designed to give ratings to English scores typed in by the user. The three types of ratings given to the various scores are as follows:

SCORE	RATING
1ØØ - 86	Excellent
85 - 61	Fair
6Ø - Ø	Poor

Therefore, English scores, which are 85 and above, will be given the rating of "excellent". Scores between 6Ø and 85 are given a rating of "fair"; and scores 59 and below are considered to be "poor". Once you have typed in the above program and executed it (by typing RUN and [↵]), MPF-III will wait indefinitely for an English score to be inputted from the keyboard.

Let's type a score of 9Ø and [↵]. As shown in the printout of the sample run above, MPF-III will print

```
SCREEN

*** LEVEL: EXCELLENT ***
```

Now let's type a score of 75 and MPF-III will respond

```
SCREEN

*** LEVEL: FAIR ***
```

since 75 is in the range from 61 through 85.
After typing in a score of 5Ø, MPF-III will then print

```
SCREEN

    *** LEVEL: POOR ***
```

To stop executing the program, press CONTROL C followed by [↵] and MPF-III will print

```
SCREEN

    BREAK IN 10
```

to signify that the execution of the program was stopped at line 10. Now let's look at how MPF-III executes this program. In line 10, the INPUT instruction tells MPF-III to print

    INPUT ENGLISH SCORE

and causes MPF-III to wait indefinitely for the user to type in a score. In line 20, there is a conditional statement to see if the score you typed in is greater then 85. If it is, then program execution is to continue with line 50. If not, then execution of the program will continue with line 30. In line 30, MPF-III will check to see if the score is less than 60 -- if the score is less than 60, then MPF-III will go to line 60; if not, then program execution will continue with the next line number. In line 40, the PRINT instruction will cause MPF-III to print

    *** LEVEL: FAIR ***

assuming that if the score is not greater than 85 and not less than 60, it must therefore be in between the range 60 and 85. In line 45, the GOTO statement will cause MPF-III to loop back to the beginning of the program (at line 10). In line 50, PRINT will instruct MPF-III to print out the

    *** LEVEL: EXCELLENT ***

message. In line 55, a GOTO statement is used to loop back to line 10. When line 60 is executed, MPF-III will print the

    *** LEVEL: POOR ***

message. Line 65 will cause MPF-III to loop back to the beginning of the program. To stop execution of this program, simply type CONTROL C and [←].

1.3.3 FOR...NEXT

The FOR ... NEXT statement enables MPF-III to perform one of its most useful functions: looping. In other words, you will be able to instruct MPF-III to repeatedly execute specific instructions through the use of the FOR...NEXT instruction. Let's now suppose that we would like to construct a multiplication table of 2's. We could accomplish this task by writing a program like the one below:

SCREEN

```
 10  PRINT "2*1=";2 * 1
 20  PRINT "2*2=";2 * 2
 30  PRINT "2*3=";2 * 3
 40  PRINT "2*4=";2 * 4
 50  PRINT "2*5=";2 * 5
 60  PRINT "2*6=";2 * 6
 70  PRINT "2*7=";2 * 7
 80  PRINT "2*8=";2 * 8
 90  PRINT "2*9=";2 * 9
100  PRINT "2*10=";2 * 10
]RUN

2*1=2
2*2=4
2*3=6
2*4=8
2*5=10
2*6=12
2*7=14
2*8=16
2*9=18
2*10=20
```

Although the above program accomplishes the task of producing a multiplication table of 2's for the numbers 1 through 10, we can easily produce the same table using the IF...THEN instruction as shown in the program below:

SCREEN

```
10 FOR I=1 TO 10
20 PRINT "2*";I;"=";2*I
30 NEXT I
40 IF I<>11 THEN 20
]RUN

2*1=2
2*2=4
2*3=6
2*4=8
2*5=10
2*6=12
2*7=14
2*8=16
2*9=18
2*10=20
```

When this program is executed, you will discover that the program output will turn out to be the same as the previous two programs. The first run of this program starting at line 10, the variable I is set to the value of 1.

Then, at line 30, the NEXT instruction will increment the current value of I by 1; therefore, the value of I will change to 2. The program will then execute line 40 where MPF-II will check to see if I <> 10. If I <> 10, then MPF-II will loop back to line 20. When NEXT is executed for the second time, the value of I will be 3 and continue on to line 40, which will cause MPF-II to loop back to line 20. FOR...NEXT will cause the program to loop until the program is executed for the 10th time and the value of I is 11; whereupon program execution will halt.

All of the previous programs are simple sample programs. Let's now look at the program below:

SCREEN

```
10  FOR I = 2 TO 10
20  FOR J = 1 TO 10
30  PRINT I;"*";J;"=";I*J
40  NEXT J
50  NEXT I
```

The program above will produce the exact same results as the preceeding programs to produce a multiplication table of 2's. Note that in the sample program above, the J-loop (inner loop) is contained within the I-loop (outer loop).

1.3.4 GOSUB...RETURN

In MBASIC, GOSUB and RETURN are another useful pair of statements which can be used to perform the same action in several different places in your program. Therefore, you can use the GOSUB and RETURN statements to avoid duplication of identical statements for the same action each time it is used within your program.

When MBASIC comes across a GOSUB statement, it branches to the line whose number follows GOSUB, and will recall where it was in the program before it branched. Refer to the shape table program in Chapter 8, Section 8.1., for a sample of the use of the GOSUB...RETURN commands.

1.3.5 READ...DATA...RESTORE

In your program, assume that you use a lot of numbers which do not change each time the program is executed, but which are simple to change if required. To take care of such tasks, MBASIC has incorporated special statements called the READ and DATA statements.

Type in the following sample program to see how the READ...DATA...RESTORE instruction can be used to read in data from a DATA statement and to place the data list pointer back to the beginning of the data list through the use of the RESTORE instruction:

SCREEN

```
10 READ A,B
20 PRINT "THE SUM OF A AND B IS "; A+B
30 READ C,D
40 PRINT "THE SUM OF C AND D IS "; C+D
50 RESTORE
60 READ E,F
70 PRINT "THE SUM OF E AND F IS "; E+F
80 DATA 1,3,5,7,9,11
90 END
]RUN

THE SUM OF A AND B IS 4
THE SUM OF C AND C IS 12
THE SUM OF E AND F IS 4
```

Upon execution of this program, MPF-III will initially read the first two data elements 1 and 3 from the DATA statement in line 80 and assign the data elements to the variables A and B, respectively. Line 20 will cause MPF-III to print

THE SUM OF A AND B IS 4

where the sum of A and B (1+3; which is 4) is also printed. In line 30, the READ instruction tells MPF-III to take the next two data items in the DATA statement and assign them to variables C and D. The PRINT instruction in line 40 will cause

THE SUM OF C AND D IS 12

to be printed on your screen. Then in line 50, note that the RESTORE instruction has been inserted here to set the data list pointer (which is currently pointing to the fifth element of the DATA statement -- since 4 elements have already been read in) back to the first element of the DATA statement. Hence, after execution of line 50, the data list pointer should now be pointing to the data element of 1. When the READ statement in line 60 is executed, the data elements 1 and 3 will be assigned to variables E and F. Therefore, when MPF-III executes line 70 to print out

```
THE SUM OF E AND F IS 4
```

You will discover that the sums of A+B and E+F are exactly the same. Line 90 simply contains an END instruction to stop the execution of the program.

1.3.6 Low-Resolution Graphics

Type

SCREEN
```
GR
```

As a result of typing GR, the first 20 lines on the computer screen will be blacked out and the remaining four lines on the bottom will consist of lines of text. MPF-III is now in its LOW-RESOLUTION "COLOR GRAPHICS" mode.

Now type

SCREEN
```
COLOR=9
```

MBASIC will respond with the prompt character

SCREEN
```
]
```

and the cursor, but intrinsically it will remember that you have selected the color green.

To draw a square using the color you have just selected, type

SCREEN
```
PLOT 20,20
```

and MBASIC will respond by displaying a small green square in the center of your screen. If the small square is not green, it means that your TV set has not been adjusted properly. Adjust the tint and color controls so that the square is green.

Now type

SCREEN

```
HLIN Ø, 39 AT 1Ø
```

MBASIC will draw a horizontal line across the leftmost three-quarters of the screen, one-quarter down from the top.

To change to a new color, type

SCREEN

```
COLOR=14
```

and then type

SCREEN

```
VLIN 1Ø,39 AT 3Ø
```

to plot a vertical line from the y-coordinate position of 1Ø to the y-coordinate position of 39 at the value of the given x-coordinate (the horizontal position) of 3Ø.

A more detailed description on color GRaphics is provided later on in this text. To return to the all text mode, type:

SCREEN

```
TEXT
```

When printing the answers to problems, it is often desirable to include text along with the answers, so as to explain the meaning of the numbers.

Now type

SCREEN

```
PRINT "ONE FIFTH IS EQUAL TO", 1/5
```

MBASIC will respond

SCREEN

```
ONE FIFTH IS EQUAL TO    .2
```

1.3.7 High-Resolution Color Graphics

If you are familiar with MPF-III's low-resolution graphics, you will discover that learning about high-resolution graphics is easy. The commands are almost the same with the exception that an H(for high-resolution graphics) is appended to the commands. For example,

HGR

is the command that is used to set the high-resolution graphics mode, clears the screen to black and leaves 4 lines for text at the bottom of the screen. In the high-resolution graphics mode, you can plot points on a 280 x-position (width) by 160 y-position (length) grid which allows you to draw with more detail than a 40 by 40 grid in the low-resolution mode. To return to the normal text mode, simply type

TEXT

Besides the HGR screen, there is a second high-resolution screen you can access by typing the command

HGR2

This command clears the screen to black and gives you a 280 x-position by 160 y-position grid with a line at

the bottom for your program. You can return to the text mode by typing

```
TEXT
```

Isn't the high-resolution mode marvelous? Of course, it is. However, you do have to give up something for this new skill. In the high-resolution mode, you have fewer colors available for use. The command to set colors in high-resolution graphics is

```
HCOLOR=C
```

in which C is a number from Ø(black) to 7(white). For plotting in high-resolution graphics, you can use the following commands:

```
HCOLOR=5
HGR
HPLOT 5Ø,8Ø
```

The command HPLOT plots a high-resolution dot in the color you have selected with the command HCOLOR (orange) at the point x=5Ø, y=8Ø. Like low-resolution graphics, x=Ø is at the leftmost top corner of the screen, increasing to the right; y=Ø is at the very top of the screen, increasing toward the bottom of the screen. The maximum value of x is 279 and the maximum value of y is 191 (however, in high-resolution mixed graphics-plus-text-mode, the maximum value of y is 159).

Now take a look at how a orange line can be drawn from specified points. Type

SCREEN

```
HPLOT 3Ø,25 TO 1ØØ,9Ø
```

As a result of executing this command, you will see a orange line drawn from the point x=3Ø, y=25 to the point x=1ØØ, y=9Ø. Through the use of the HPLOT command, you will have the ability to draw lines between any two points on the screen. Now let's try to connect another line to the end of the previous one. Type the following command

SCREEN

```
HPLOT TO 20,65
```

In this example above, this HPLOT command uses the last point previously plotted as its starting point and uses the same color from that point to draw a line to the specified point after TO (in this case, to the point x=20, y=65). Now try the following command and see what happens.

SCREEN

```
HPLOT 0,0 TO 279,0 TO 279,159 TO 0,159 TO 0,0
```

After execution of this command, you should have a white border around all four sides of the screen displayed.

While operating in the graphics mode, VTAB and HTAB are commands which may be used in your program to move the cursor and to print a character at a predetermined position on your screen. Take a look at the following commands and their actions:

COMMAND	ACTION
VTAB 1	sets the cursor at the top line
VTAB 24	sets the cursor at the bottom line
HTAB 1	sets the cursor at the leftmost position of the current line
HTAB 40	sets the cursor at the rightmost position of the current line

For more details on VTAB and HTAB, refer to Chapter 4.

If you want to return to programming in the text mode, type CONTROL C followed by also typing

```
SCREEN

    TEXT
```

and you will be in the text mode.

CHAPTER 2
COMMANDS PERTAINING TO SYSTEM AND UTILITY

In the preceding chapter, frequently encountered commands and syntactic definitions were introduced. Now that you have been provided with some of the basic ground rules to MBASIC programming, this chapter will discuss commands related to the system and utility of your MPF-III. In other words, these commands will allow you to communicate to MPF-III that you want to access a machine language subroutine, display the contents of a particular memory location, store data in a specific memory location and so on.

2.1 CALL

```
Execution mode: imm & def
        Format: CALL aexpr
```

When you use the CALL instruction, you will cause the computer to execute a machine-language subroutine whose memory location is specified by the argument \aexpr\.

If \aexpr\ is not in the range -65535 through 65535, then you will receive the

```
SCREEN
    ?ILLEGAL QUANTITY ERROR
```

message. In fact, \aexpr\ is restricted to the range of addresses for which valid memory devices exist. (Refer to HIMEM: and POKE for more details.)

Note that positive and negative addresses which are equivalent to each other may be used alternately. For example, "CALL -936" and "CALL 64600" are the same.

Several examples of the use of the CALL instruction are contained in Appendix E of this manual.

2.2 HIMEM:

```
Execution mode: imm & def
        Format: HIMEM: aexpr
```

The HIMEM: instruction establishes the address of the highest memory location available to a BASIC program, including variables. As you may have noticed, HIMEM: is an abbreviated version of HIgh MEMory. By using HIMEM:, you can protect the area of memory for data, graphics or machine language routines. In order to avoid receiving the

```
SCREEN
    ?ILLEGAL QUANTITY ERROR
```

message, \aexpr\ needs to be inclusively in the range -65535 through 65535.

Be aware that unless there is appropriate memory hardware at the locations specified by all addresses up to and including \aexpr\, your programs may not execute properly.

Generally speaking, the maximum value of aexpr is ascertained by the amount of memory available in the computer. Since the MPF-III has 64K memory, the \aexpr\ could be as high as 65535.

When MBASIC is initiated, it automatically sets HIMEM: to the highest memory address available on your computer. The present value of HIMEM: is stored in memory locations 116 and 115 (decimal). If you now type

SCREEN

```
PRINT PEEK(116)*256+PEEK(115)
```

the computer will display the current value of HIMEM: stored in memory.

If the value of HIMEM: is lower than the value of the memory address set by LOMEM:, or when the memory available left by HIMEM: is insufficient for the program to execute, the following error message will be displayed on your screen.

SCREEN

```
?OUT OF MEMORY ERROR
```

Observe that because equivalent positive and negative values for \aexpr\ are interchangeable, the value of \aexpr\ may be inclusively in the range of 0 through 65535, or in the range of -65535 through -1.

CLEAR, RUN, NEW, DEL, RESET or by making changes or additions to a program line, will not reset HIMEM: in memory.

2.3 LOAD and SAVE

Execution mode: imm & def
Format: LOAD
        SAVE

LOAD is an instruction to read in a program stored on a cassette tape into the computer.

When you have connected the MPF-III to a tape recorder in order to read in a program. You can follow the steps described below to perform a read from tape operation.

1) Press L O A D on the MPF-III keyboard.
2) Press the PLAY button on the cassette tape recorder.
3) Press the carriage return key on the MPF-III keyboard. After the carriage return key is pressed, the computer will begin the read operation. When the computer begins the read (load) operation, it will beep. When the computer finishes the read operation, it will beep again. If an error occurs during the load operation, the computer will print an error message on the screen or continue to read the next file on the cassette tape.

SAVE is an instruction to store a program onto a cassette tape.

When you have connected the MPF-III to a tape recorder in order to save a program. You can follow the steps described below to perform a save to tape operation.

1) Press S A V E on the MPF-III keyboard.
2) Press the RECORD and PLAY buttons on the cassette tape recorder.
3) Press the carriage return key on the MPF-III to start the SAVE operation. When the computer begins the SAVE operation, it will beep. When the computer finishes the SAVE operation, it will beep again.

To interrupt the execution of a LOAD or SAVE command, simply press RESET.

If the first characters of a variable name start with LOAD or SAVE, the command will be executed first before you receive the error message.

SCREEN

```
?SYNTAX ERROR
```

message. For example, if you type the following statement into the computer,

SCREEN

```
SAVESETTING=9
```

MBASIC will first attempt to SAVE the current program on cassette tape before it gives you the .

SCREEN

```
?SYNTAX ERROR
```

message.

2.4 LOMEM:

Execution mode: imm & def
Format: LOMEM: aexpr

The LOMEM: instruction assigns the address of the lowest memory location available to a BASIC program.

The address of the lowest memory location available is normally the address of the starting memory location for the first BASIC variable. Before execution of the program, MBASIC automatically sets LOMEM: to the end of the program. By using the LOMEM: statement, you can protect variables in safe areas of memory set aside from high-resolution graphics in computers with large amounts of memory.

Note that \aexpr\ must be inclusively in the range -65535 through 65535; otherwise, the computer will display the following error message

SCREEN

```
?ILLEGAL QUANTITY ERROR
```

If the value of LOMEM: is assigned a value which is higher than HIMEM: then the message below will appear.

SCREEN

```
?OUT OF MEMORY ERROR
```

In other words, \aexpr\ must be lower than the greatest possible value set by the HIMEM: statement. However, if the value of LOMEM: is set lower than the address of the highest memory location occupied by the current operating system(including any currently stored program), you will receive the same error message shown below.

SCREEN

```
?OUT OF MEMORY ERROR
```

If you want to reset the LOMEM: statement, use NEW, DELETE or make changes or additions to a program line. However, be aware that such commands as RUN, typing the RESET key followed by a CONTROL C(holding down the CONTROL key while typing C) and hitting the RETURN key or typing the RESET key and then hitting the RETURN key will NOT reset the LOMEM: statement.

The present value of LOMEM: is stored in memory locations 1Ø6 and 1Ø5. If you would like to see the current value of LOMEM:, then type

SCREEN

```
PRINT PEEK(1Ø6)*256+PEEK(1Ø5)
```

After LOMEM: is set, it is legal to use one of the above commands to reset the LOMEM: statement; however, if the new value is lower than the current value of LOMEM:, the following error message will be generated because the new value of LOMEM: can only be set to a higher value.

SCREEN

```
?OUT OF MEMORY ERROR
```

If LOMEM: is changed during the execution of a program, then it is possible that certain stacks or portions of the program may not be available and the program will not be able to continue to execute correctly.

Note that the equivalent positive and negative values for \aexpr\ are interchangeable.

2.5 NEW

Execution mode: imm & def
Format: NEW

The NEW instruction has no parameters. You can use NEW to completely delete your program currently in the RAM and all variables. This instruction is normally used just before typing in a new program and also to get rid of the current program.

2.6 PEEK

Execution mode: imm & def
Format: PEEK(aexpr)

This instruction can be used to return the contents, expressed in decimal, of the byte at address \aexpr\. Some examples of how to use PEEK are in Appendix E of this manual.

2.7 POKE

Execution mode: imm & def
Format: POKE aexpr1, aexpr2

The POKE instruction can be used to store the binary equivalent of the decimal value \aexpr2\ into the \aexpr1\ location in memory. The range of \aexpr2\ has to be from Ø through 255, and \aexpr1\ must be from -65535 through 65535. The use of reals with this command are first converted to integers before execution can begin. If the values are out of range, you will receive the following error message:

SCREEN

```
?ILLEGAL QUANTITY ERROR
```

\aexpr2\ will not be stored properly at non-receptive addresses; such as the Monitor ROM, unless the address specified by \aexpr1\ has the appropriate receiving hardware(memory, or a suitable output device).

Generally speaking, the range of \aexpr1\ can be from Ø through maximum amount of memory in the computer.

In the computer's memory, there are many locations which contain information essential to the operations of the computer system. By using the POKE instruction to access these locations, you can easily change the operation of the system, of your program, and more seriously, change the system. So we suggest that you be very careful when you use the POKE instruction.

2.8 RUN

Execution mode: imm & def
Format: RUN [linenum]

When you type RUN, the computer will start executing your program at the line number indicated by linenum and will clear all variables, pointers, and stacks. If you did not specify a line number from which you want to start program execution, RUN will start at the lowest numbered line in the program. If there is no program in memory then the computer will return control to you(the user).

If linenum is given in deferred-execution mode but does not already exist in the program, or if linenum is negative, the computer will display the following message:

SCREEN

```
?UNDEF'D STATEMENT ERROR
```

Moreover, if the value of linenum is greater than 63999, then the error message

SCREEN

```
?SYNTAX ERROR
```

will be displayed with no indication in which line the error occurred.

In the deferred-execution mode, however, the above error messages will change to the following:

```
?UNDEF'D STATEMENT ERROR IN XXXX  and
?SYNTAX ERROR IN XXXX
```

where XXXX can be different line numbers, normally above 65000.

2.9 STOP, END, CONTROL C, RESET and CONT

Execution mode: imm & def
Format: STOP END CONT

Execution mode: imm only
Format: CONTROL C reset

The STOP instruction can be used to stop program execution, and return control to the user. When you issue a STOP instruction to the computer, it will print

```
SCREEN

    BREAK IN linenum
```

where linenum is the line number of the program statement which performed the STOP.

Like STOP, the END instruction will cause program execution to cease; however, no error message will be printed.

The effect of a CONTROL C is the same as the insertion of a STOP statement immediately after the statement that is presently being run and can be issued as a command by pressing the CONTROL key and the letter "C" key on your keyboard simultaneously. You can also use CONTROL C to interrupt an MBASIC LISTing and an INPUT instruction; however, an INPUT is interrupted only if CONTROL C is the first character entered and if the RETURN key [←┘] is pressed.

By using reset, you can immediately stop the execution of a program or command and can perform a reset by pressing the RESET key on your keyboard. Although you will not lose your program, it is possible that some pointers or stacks will be cleared upon execution of a reset. After typing reset, you will be in the system monitor program as indicated by the prompt character (@). If you want to return to MBASIC without destroying the current program, just type CONTROL C and the return key [⟵].

If STOP, END or CONTROL C was used to cease program execution, the CONT command can be used to instruct the computer to CONTinue execution at the next instruction and not the next line number since one line may contain several instructions. If no program was stopped, then the CONT command has no effect. After RESET or CONTROL C return the program, it may not CONTinue to run properly, because some pointers and stacks may have already been cleared.

If an INPUT statement is stopped by CONTROL C and you attempt to CONTinue execution of the program by typing CONT, you will receive the following error message:

SCREEN

```
?SYNTAX ERROR IN linenum
```

in which linenum represents the line number of the INPUT statement.

If you type CONT and the computer responds with the

SCREEN

```
?CAN'T CONTINUE ERROR
```

message, then this means after execution of the program ceases, the user may have

(a) changed or deleted a program line, or
(b) tried to perform an operation which results in an error message.

In using immediate-execution mode commands, the program variables can be changed as long as no error messages were given.

If you use the DEL instruction in a deferred-execution statement, the specified lines are struck out and then program execution is stopped. If you try to continue execution of the program by typing CONT in this instance, you will receive the

SCREEN

```
?CAN'T CONTINUE ERROR
```

message.

If you use CONT in a deferred-execution statement and program execution is stopped at that statement, the control of the computer is not returned to the user. You can get control of the computer back again by giving a CONTROL C command. Now if you try to continue program execution by typing CONT, control is given to the halted program and it will continue to execute.

2.10 TRACE and NOTRACE

Execution mode: imm & def
Format: TRACE
NOTRACE

The command TRACE can be used to set the computer to a debug mode which will display the line number of each statement as it is run during program execution. As the program is being printed on the screen; TRACEs may be shown in an peculiar manner or overwritten. The command NOTRACE, the opposite of TRACE, turns off the TRACE debug mode. (Note: Once a TRACE has been set, it cannot be turned off by typing RUN, CLEAR, NEW, DEL or RESET.)

2.11 USR

Execution mode: imm & def
Format: USR (aexpr)

This function, USR, passes the argument \aexpr\ to a machine-language routine. After aexpr is evaluated and placed into a floating point accumulator (locations $9D through $A3), a JSR to location $0A is executed. Locations $0A through $0C must have a JMP to the beginning of the machine-language subroutine. The return value for the function is stored in the floating point accumulator.

To get a 2-byte integer from the floating-point accumulator, your subroutine must perform a JSR to $E10C. The integer value will be in locations $A0(high-order byte) and $A1(low-order byte) when it is returned.

To transform an integer result to its equivalent value in floating-point form, put the two-byte integer in registers A (high-order byte) and Y(low-order byte). After you have accomplished this, do a JSR to $E2F2, and upon hitting a return, the floating point value will have been placed in the floating-point accumulator.

If you want to return to MBASIC, perform an RTS.

The simple program below will show you how the USR function is used and its format.

SCREEN

```
] RESET CALL-159 [⟵┘]
* OA:4C 00 03 [⟵┘]
* 0300:60 [⟵┘]
* CONTROL C [⟵┘]
] PRINT USR(8)*3 [⟵┘]
24
```

In the above program at location $ØA, we have set a JMP(code 4C) to location $3ØØ(low-order byte first, then high-order byte). At location $3ØØ, we have set an RTS(code 6Ø). When USR(8) is executed in MBASIC, the argument 8 was placed in the accumulator and the Monitor performed a JSR to location $ØA where it discovered a JMP to $3ØØ. In location $3ØØ, the Monitor discovered an RTS which sent it back to MBASIC. The value returned was the original value in the accumulator(in this case, the value was 8) which was multiplied by 3 to get 24.

2.12 WAIT

Execution mode: imm & def
Format: WAIT aexpr1, aexpr2 [,aexpr3]

The WAIT command permits the user to place a conditional pause into a program and can only be terminated by RESET.

In this command, the parameter \aexpr1\ is the address of a memory location which has to be in the range -65535 and 65535 to avoid receiving the

```
SCREEN

    ?ILLEGAL QUANTITY ERROR
```

message. As a matter of fact, the range of \aexpr1\ and POKE are the same; that is, from Ø through the maximum value of HIMEM: (refer to POKE or HIMEM: for more details). Of course, equivalent positive and negative addresses may be used.

The other two parameters, \aexpr2\ and \aexpr3\ used in this command, are decimals which are required to be in the range Ø through 255. As soon as the WAIT command is executed, the values of \aexpr2\ and \aexpr3\ are transformed to binary numbers ranging from Ø through 11111111.

Assume that only aexpr1 and aexpr2 are designated. In this instance, each of the eight bits in the binary contents of location \aexpr1\ is ANDed with the corresponding bit in the binary equivalent of \aexpr2\.

Therefore, this gives a zero for each bit unless both of the corresponding bits are high(1). In the event that the consequences of this process are eight zeroes, the test is performed again.

The WAIT is completed when the end result of this process is a non-zero; that is, at least one high(1) bit in \aexpr2\ was matched by a corresponding high(1) bit at \aexpr1\. After WAIT has been executed, program execution will continue with the next instruction.

If you type

SCREEN

```
WAIT aexpr1,7
```

MPF-III will cause your program to pause until at least one of the three rightmost bits at \aexpr1\ is high(1).

By typing

SCREEN

```
WAIT aexpr1,0
```

however, you have just caused the program to pause interminably, because any bit ANDed with 0 has a value of 0.

If aexpr1, aexpr2, and aexpr3 are specified, then WAIT operates as described below:

(1) Each bit in the binary contents of \aexpr1\ is XORed with the corresponding bit in the binary contents of \aexpr3\.
If the bit in \aexpr3\ is high(1), then the result is the REVERSE of the corresponding bit at \aexpr1\ (i.e., a 1 is changed to a 0 and vice-versa).
If the bit in \aexpr3\ is low(0), then the result is the SAME as the corresponding bit at \aexpr1\ and is ANDed with \aexpr2\.
If the bit in \aexpr3\ is zero, then the XOR portion is not performed.

(2) After the XOR portion is executed, each result is ANDed with the corresponding bit in the binary equivalent of \aexpr2\.
If the end result is eight zeroes, then the test is performed again. If the result is non-zero, then WAIT is completed and program execution resumes at the next instruction.

A different way to look at WAIT is as follows:

The objective of this command is to examine the contents of \aexpr1\ and see when any one of certain bits is high or any one of certain other bits is low (Ø or off). A 1 in each of the eight bits in the binary equivalent of \aexpr2\ show that you are interested in the corresponding bit at \aexpr1\, a Ø means you are not interested and want to disregard that bit. Each of the eight bits in the binary equivalent of \aexpr3\ show which state you are WAITing for the corresponding bit in \aexpr1\ to be in: a 1 means the bit must be low, a Ø means the bit must be high.

The WAIT is terminated over when any one of the bits in which you have indicated interest (by a 1 in the corresponding bit of \aexpr2\) matches the state you specified for that bit (by the corresponding bit of \aexpr3\). If aexpr3 is absent, then its default value is zero. For example:

WAIT aexpr1, 255,Ø	signifies you want to cause the program execution to temporarily stop until at least one of the 8 bits at \aexpr1\ is high.
WAIT aexpr1, 255	same as described above.
WAIT aexpr1, 255, 255	means to stop until at least one of 8 bits at \aexpr1\ is low.
WAIT aexpr1, 1, 1	means to stop until the rightmost bit at \aexpr1\ is low.
WAIT aexpr1, 3, 2	means to stop until the rightmost bit at \aexpr1\ is 1, and/or the next-to-rightmost bit is Ø.

CHAPTER 3
COMMANDS RELATING TO INPUT/OUTPUT

The more you become acquainted with your MPF-III and its functions, the more it will become apparent to you that a computer has primarily three operations:

(1) Accepting input of data,
(2) Processing or manipulating data, and
(3) Printing output (or results).

The commands discussed in this chapter will deal with the input and output operations of your MPF-III. Upon completion of this chapter, you will know how to use the various commands to get MPF-III to accept data and print it out.

3.1 DATA

Execution mode: def only
Format: DATA [literal|string|real|integer] [{,[literal|string|real|iteger]}]

DATA statements create and contain a list of elements which are used by READ statements. If the list of elements is too long to be contained in a data statement such as at instruction line number 200 (see example below), then the list of elements may be contained in another DATA statement at the next line of instruction (at line 210).

EXAMPLE:

LIST OF ELEMENTS	CLASSIFICATION
INDEX E15	string
123.61	real
BARBARA	literal
31760	real
-89.50	integer
NUMBER 186	string
DEPT 2	string

DATA STATEMENT FORMAT IN A PROGRAM WILL BE AS FOLLOWS:

SCREEN

```
200 DATA BARBARA, "INDEX E15", "NUMBER 186"
210 DATA "DEPT 2", 123.61, 31760, -89.50
```

Data statements are read by the READ statement starting from the data statement with the lowest line number. Therefore, the elements in the data statement at program line 200 in the example above must be READ first before the elements in line 210 can be READ.

In computer programs, data statements do not necessarily have to come before READ statements. In other words, DATA statements may be placed before or after READ statements anywhere in the program.

The rules for DATA elements which are READ into arithmetic variables are the same as the rules for

INPUT responses assigned to arithmetic variables. However, there is one exception in that the colon cannot be used as a character in a numeric DATA element. (Refer to section 3.5 of this chapter for more information on INPUT.)

The rules for DATA elements which are READ into string variables are the same as for INPUT responses assigned to string variables except that if CONTROL C is used as a data element, the program will not be interrupted. However, if CONTROL C is used as a data element in a numerical data list, then the

```
SCREEN

    ?SYNTAX ERROR
```

message will be displayed on your screen. Listed below are some rules which apply to string variables:

(1) Strings and/or literals may be used.
(2) Any spaces or after a string are ignored.
(3) If any quotation marks are used within a string the

```
SCREEN

    ?SYNTAX ERROR
```

message will appear on your screen. However, all other characters (including the colon and the comma, but not including CONTROL X and CONTROL M) used in that string are considered to be legal.
(4) If a literal is an element in a DATA statement, then the quotation mark is regarded as a valid character anywhere in the literal with the exception that it cannot be used as the first non-space character. The colon, the comma, CONTROL X and CONTROL M are not accepted as valid characters.

For more details regarding the rules for READing in string variables, refer to INPUT (section 3.5 of this chapter).

Acceptable DATA elements may be any combination of reals, integers, strings and literals. If you try to

use the READ statement to assign a DATA element which is a string or a literal to an arithmetic variable, the

SCREEN
```
?SYNTAX ERROR
```

message will be given with reference to that particular DATA line.

Depending on the variable to which the element is assigned, a zero(numeric) or the null string will be returned for that element if the list of elements in a DATA statement has an element which does not exist. Such elements occur in a DATA statement when any of the four following situations are true:

(1) A non-space character does not exist between DATA and return.
(2) The first non-space character following DATA is a comma.
(3) A non-space character does not exist between two commas.
(4) The last non-space character before return is a comma.

Therefore, when the following statement

```
120 DATA ,,
```

is used by a READ statement, then up to three elements consisting of zeros or null strings may be returned.

If DATA is used in the immediate-execution mode, the

SCREEN
```
?SYNTAX ERROR
```

message will not be given. However, the data elements contained in the DATA statement cannot be read by the READ statement.

3.2 DEF FN

```
Execution mode: def only
        Format: DEF FN name(real avar) = aexprl FN

Execution mode: imm & def
        Format: FN name(aexpr2)
```

The DEF FN statement allows functions to be defined and used in MBASIC programs. To define a function in a program, first use the DEF statement to define the function FN name. After execution of the program line which DEFines the function, the function may then be used in the form FN name (aexpr2) where the argument enclosed in parentheses (in this instance, aexpr2) may represent any arithmetic expression. The DEF statement defining aexprl may be only one program line in length.

During a program, such functions as described above may be reDEFined. The rules pertaining to the use of arithmetic variables in MBASIC must still be followed for your program to execute properly. Most importantly, the first two characters of the function FN name must be unique. For instance, if the following lines

```
50  DEF FN JNC(A) = SEC(A)
60  DEF FN JNS(A) = CSC(A)
```

are executed, MBASIC will interpret line 50 as the definition of an FN JN function and line 60 as the redefinition of the FN JN function.

In a DEF FN statement, real avar is known to be a dummy variable, which can be called with an argument aexpr2 when the function FN name is used later in the program. Therefore, the argument aexpr2 replaces real avar anytime it appears in the definition's aexprl. aexprl may consist of any number of variables, however, only one of the variables can correspond to real avar and hence, can correspond to the argument variable.

It is optional for real avar of the DEF FN statement to appear in aexprl. If the DEFinition's real avar does not appear in aexprl, then the function's argument is ignored in evaluating aexprl when the function is used later in the program. In spite of this, keep in mind that the argument of the function is evaluated, so it must be legal. Take the following program as an example:

SCREEN

```
10 DEF FN A(M) = 3*M+M
20 PRINT FN A(18)
30 DEF FN B(N) = 9+4
40 X = FN B(18)
50 PRINT X
60 DEF FN A(Q) = FN B(R) + Q
70 PRINT FN A(X)

RUN
72          [FN A(18) = 3*18+18]
13          [FN B(anything) = 13]
26          [new FN A(13) = 13+13]
```

The following error message

SCREEN

```
?UNDEF'D FUNCTION ERROR
```

will appear on your screen if a deferred-execution DEF FN name statement is not executed before FN name is used.

The use of string functions defined by the user are not permitted. The use of an integer name% for name or for real avar to define functions are also not permitted.

The definition of a new function by a DEF statement takes up six bytes in memory to store the pointer to the definition.

3.3 GET

Execution mode: def only
Format: GET var

The GET instruction takes a character from the keyboard without displaying it on the TV screen and does not require the user to press the RETURN [←] key.

When GET is used with the string variable (that is,

when GET svar is used), then the execution of the program will stop if CONTROL C is typed.

Although MPF-III was not designed to take in values for arithmetic expressions,

```
GET avar
```

may be used under the following limitations:

- An attempt to GET a comma or a colon will cause the

SCREEN

```
?EXTRA IGNORED
```

message to appear on your screen.

- The period (.), plus sign (+), minus sign (-) and any spaces will have the same result as if you had typed in a zero and will return a zero.

- Pressing the RETURN [←┘] key, non-numeric keys or CONTROL @ will result in causing the

SCREEN

```
?SYNTAX ERROR
```

message to be presented.

- When ONERR GOTO...RESUME is used and if two consecutive GET errors occur, then the system will "hang" until the RESET key is pressed. If GOTO is used in place of RESUME, then the program will be executed untl the 43rd GET error (which may be in any order) occurs, at which point the program will jump to the Monitor. In order to regain control of the computer, press RESET CONTROL C and the RETURN [←┘] key.

Due to the limitations discussed above, you are recommended to use

GET svar

to GET numbers and later convert the value of the resulting variable to a number by using the VAL function.

3.4 INPUT

Execution mode: def only
Format: INPUT [string;] var[{,var}]

Upon execution of an INPUT statement, MPF-III will wait until it receives input from the keyboard. If the optional string in the command format above IS NOT given, then the INPUT statement causes MPF-II to print a question mark and wait for you to type one of the following:

(1) a number, if var is an arithmetic variable, or
(2) characters, if var is a string variable.

After receiving input from the keyboard, MPF-III takes the value of the number or string and stores it in var.

If the optional string IS given, it is returned and printed exactly as designated by the user. Spaces, question marks and punctuation marks will not be printed after the string. As shown in the command format above, only one optional string can be specified in an INPUT statement. If a string is used, make sure it is placed immediately after "INPUT" and followed by a semicolon(;).

Only a real or an integer will be accepted by an INPUT statement. The use of arithmetic expressions with

INPUT is not legal. The space, plus sign (+), minus sign (-), the period (.), and E (which are all considered to be characters) are acceptable as numeric input. If these characters or any combination of these characters are used in the appropriate form, then it will be accepted by INPUT. For example, the concatenation of the following characters in the form: +E- wll be accepted by INPUT and be evaluated as zero. However, concatenation of the characters +- will not be accepted by INPUT.

Although the use of any spaces are acceptable as numeric input, they are generally ignored. The use of numeric input which is not a real, an integer, a comma or a colon will cause the message

SCREEN

```
?REENTER
```

to appear on the screen and also cause the INPUT instruction to be executed again.

In the same way that INPUT considers an arithmetic expression unacceptable as numeric input, it also considers a string expression to be unacceptable as a string variable. A response given to a string variable is acceptable if it is a single string or a literal. Any spaces which precede the first character are ignored.

IF THE RESPONSE TO A STRING VARIABLE IS A SINGLE STRING, then all characters within the string can be any character (including the colon and the comma) except for the quotation mark, CONTROL X and CONTROL M. Any spaces placed after the final quotation mark are ignored.

IF THE RESPONSE TO A STRING VARIABLE IS A LITERAL, then all characters within the string can also be any character except for the comma, the colon, CONTROL X and CONTROL M. Quotation marks are acceptable as characters and can be contained in any portion of the literal with the exception of the first non-space character. Any spaces placed after the literal are acceptable and are considered to be part of the literal.

If an INPUT instruction expected to receive NUMERIC

INPUT but instead received a RETURN ←┘, then the

```
SCREEN
    ?REENTER
```

message will be displayed and cause the INPUT instruction to be executed again. On the other hand, if INPUT expected to receive a STRING RESPONSE, but instead received a RETURN [←┘], then the response is considered to be a null string and the execution of the program will resume.

Many variables can be contained in a single INPUT instruction. Although a mixture of both string variables and arithmetic variables in the same INPUT instruction is allowed, the response you type in must correspond to the type of input the INPUT instruction expects to receive. In other words, if INPUT expects to receive a string response, then your response should be a single string or a literal. Similarly, if INPUT expects to receive an arithmetic variable, then the numbers you type in must conform to the rules for numeric input (which was previously discussed in this section). The response you type in may be separated by commas or RETURNs [←┘]. Consequently, commas are considered to be response separators if a response that does not start with a quotation mark is typed in. This only applies when INPUT expects to receive a single response.

A colon that is typed in an INPUT response which does not begin with a quotation mark causes all the characters (including commas) typed in after the colon to be ignored. Therefore, the INPUT instruction will accept the next response which is first preceded by a RETURN [←┘].

Typing a RETURN [←┘] before all the var's have been given responses will cause two question marks to be printed; indicating that it expects to receive an additional response. However, if the response typed in exceeds the expected number of var's in an INPUT instruction, or if there is a colon in the final expected response (not contained in a string), then the

SCREEN

```
?EXTRA IGNORED
```

message is printed and execution of the program will resume.

If the first character of an INPUT response is a colon or a comma, the response is considered to be a zero or a null string. However, in the INPUT instruction, commas are required to separate variables and a semi-colon must be placed immediately after an optional string. Therefore, it is important to pay attention to which punctuation marks can or cannot be used in INPUT statements and responses.

If CONTROL C is the first character typed in an INPUT instruction, then it will be able to interrupt the INPUT instruction. When a RETURN ⟵┘ is typed, then the execution of the program is stopped. If the user tries to CONTinue program execution in this instance, the message

SCREEN

```
?SYNTAX ERROR
```

will be given. If CONTROL C is not the first character typed in an INPUT instruction, then it is regarded to be like any other character.

An attempt to use the INPUT statement in the immediate-execution mode will result in the presentation of the

SCREEN

```
?ILLEGAL DIRECT ERROR
```

message on the screen.

3.5 LET

Execution mode: imm & def
Format: [LET] avar [subscript] = aexpr
[LET] svar [subscript] = sexpr

This statement assigns the value of the expression or string on the right-hand side of the "=" sign to the variable name on the left. As exemplified in the example below, typing the word LET is optional in assigning values to variables.

EXAMPLE:

LET B=1Ø is the same as **B=1Ø.**

If any one of the three situations below exist:

(1) an attempt is made to assign an arithmetic expression to a string variable name, or
(2) a literal is assigned to a string variable name, or
(3) a string expression is assigned to an arithmetic variable name,

then the

SCREEN

```
?TYPE MISMATCH ERROR
```

message will be presented on your screen.

On the other hand, if an attempt is made to assign a literal to an arithmetic variable name, then MBASIC tries to interpret the literal as an arithmetic expression.

3.6 PRINT

Execution mode: imm & def
Format: PRINT [{expr}[{,|;[{expr}]}]][,|;]
PRINT {;}
PRINT {,}

The PRINT command is used to output characters to the

screen or other output devices. If you simply type

SCREEN
```
PRINT
```

without any options, then a line feed and a return will be outputted. However, if PRINT is used with options, the value of the expressions contained in the PRINT statement will be printed. The appearance of the values depends on the type of expressions to be printed and on where semicolons or commas have been used in between the values. Spacing between printed values are determined by the use of semicolons or commas in a PRINT statement. If semicolons are used, then the next value to be printed is PRINTed immediately following the last value printed. If commas are used, then the next value to be printed is PRINTed in the first positin of the next available tab field.

In MBASIC, tab fields are spaced 16 characters apart; tab field 1 begins at column 1 and ends at column 16; tab field 2 starts at column 17 and ends at column 32; and tab field 3 starts at column 33 and ends at column 4Ø. Tab field 2 is only available for printing if nothing is printed in column 16. Likewise, tab field 3 is available for printing only if nothing is printed in columns 24 through 32 as shown below.

TAB FIELD 1		TAB FIELD 2	TAB FIELD 3
1	1617	3233	4Ø
*starts in column 1	*starts in column 17 *available if nothing is printed in column 16		*starts in column 33 *available if nothing is printed in columns 24-32

If the list of expressions in a PRINT statement does not end with either a comma or a semicolon, MPF-III will automatically insert a line feed and a return after the last item on the list. A list of expressions in a PRINT statement which ends with a comma will print the next output in the first position of the next available tab field. If a list of expressions ends with a semicolon, then the item to be printed by the next PRINT statement will be printed immediately after the last character of the item printed by the most recently executed PRINT statement without any spaces inserted between the two items.

By using some of the POKE commands contained in Appendix E of this manual, you will be able to change the size of the scrolling window for text. You may want to change the size of your scrolling window if PRINT tab field 3 is not working properly (usually happens when the text window is less than 33 positions wide). Instead of resetting the text window, you may also use HTAB to PRINT a first character outside of the text window.

When items are listed in a PRINT statement without semicolons or commas, the items are concatenated if the items can be interpreted without any syntax problems. For example, if the following were given:

```
A=5 : B=6 : C=7 : D=8 : D4=9
```

and used in the following PRINT statements, the items will be printed as shown below:

SCREEN

```
PRINT 1/6(2*1)49                 :PRINT 5(A)6(B)7(C)8(D)D4
Ø.166666666249                    556677889

PRINT 1.2.3.4.5. ,               :PRINT C."D."D.6
1.2.3.4.5Ø                        7ØD.8.6
```

If periods are used in PRINT statements and cannot be interpreted as a decimal point as shown in the example above, it will be treated as the number zero (Ø). Although a list of semicolons following a PRINT instruction is legal, it will have the same effect as just using PRINT. A list of commas following PRINT will cause spacing over to the next available tab field

to occur for each comma (with a limit of 239 characters per instruction).

If the length of two concatenated strings (S$ and T$ in the example below) is greater than 255 characters long, then an attempt to execute

SCREEN

```
PRINT S$ + T$
```

will produce the

SCREEN

```
?STRING TOO LONG ERROR
```

message to appear on your screen. The following concatenation,

SCREEN

```
PRINT S$ T$
```

however, is legal and can be executed regardless of its length. If desired, you can use the question mark (?) as an abbreviated form of the PRINT command to save time typing. Although it is displayed as a question mark when you type it in, it will be shown as "PRINT" when the program is listed again.

3.7 READAD

Execution mode: imm & def
Format: READ var [{,var}]

Upon execution of a program, the value of the first variable in the first READ statement will take on the value of the first element contained in the DATA list, which is composed of all the elements in all the DATA statements. If there is a second variable, it will be assigned the value of the second element in the DATA list; the third variable will take on the value of the third element and so on. After it is executed, the READ statement then leaves a data list pointer after the most recently used data element. The data list pointer acts as a reminder of which element was READ last so that if there is another READ statement to be executed later in the program, then the next element following the last element used will be READ. If you would like to set the pointer back to the first element in the DATA list, use RUN or RESTORE.

Trying to READ more data than the data contained in the DATA list will cause the

SCREEN

```
?OUT OF DATA ERROR IN linenum
```

message to be printed; in which linenum represents the line number of the offending READ statement that requested additional data.

On the other hand, additional data which is not used in the DATA list by READ statements in a program will not cause an error message to be displayed and is acceptable.

Although it is not necessary for a stored program to be executed before elements from DATA statements can be READ, they can only be READ if they already exist as lines in a stored program in the immediate-execution mode. If the DATA statement does not exist in a stored program, the

SCREEN

```
?OUT OF DATA ERROR
```

message is displayed. The execution of a program in the immediate-execution mode will NOT place the pointer at the beginning of the DATA list.

3.8 RESTORE

Execution mode: imm & def
Format: RESTORE

The use of the RESTORE command will move the data list pointer (refer to the READ and DATA statements) to the beginning of the data list. This command has no parameters or options.

3.9 PR#

Execution Mode: imm & def
Format: PR# aexpr

The PR# command is used to send the output to slot aexpr. aexpr must be in the range from 0 through 7.

PR# 0 sends the output to screen rather than any slot. If aexpr is less than 0 or greater than 255, the error message

?ILLEGAL QUANTITY ERROR

will be output on the screen.

PR#1 outputs to the printer.
PR#4 outputs to the screen in Chinese Mode.
PR#3 causes the screen to enter 80-column mode.

3.10 IN#

Execution mode: imm & def
Format: IN# aexpr

IN# instructs the MPF-III to fetch input from aexpr (slot #). aexpr is an integer in the range from 1 to 7.

IN#0 tells the MPF-III to fetch input from the keyboard instead of from other peripheral devices.

If aexpr is less than 0 or greater than 255, then the following error messages will be output on the screen.

?ILLEGAL QUANTITY ERROR

CHAPTER 4
COMMANDS CONCERNING EDITING AND FORMAT

While typing in some of the programs in the previous chapters, you may have used the left and right arrow keys for correcting typographical errors. You may have also remembered vaguely from the Introduction to Micro-Professor III BASIC Programming Manual that you can use the LIST function to list the program to verify that you have typed in deferred-execution mode instructions correctly. Other such commands for program editing are presented in this chapter. In addition to program editing commands, commands pertaining to formatting and cursor movement have also been incorporated into this chapter. You will find that these commands are especially helpful in presenting your program output in the desired format.

4.1 CLEAR

Execution mode: imm & def
Format: CLEAR

The CLEAR command has no parameters and clears all arrays, variables, and strings. It also resets all pointers and stacks.

4.2 CONTROL X

Execution mode: imm only
Format: CONTROL X

CONTROL X instructs MPF-III not to pay any attention to the current line being typed, without omitting the previous line of the same line number. At the end of the line to be ignored, a backslash(\) will be displayed and the cursor will go to column Ø of the next line. CONTROL X can also be given during a response to an INPUT instruction if the typed response is to be ignored.

4.3 DEL

Execution mode: imm & def
Format: DEL linenum1, linenum2

The DEL command will delete the lines from linenum1 through linenum2, inclusive. If linenum1 is not a program line number currently contained in the program, the next larger line number in the program is used instead of linenum1. Likewise, if linenum2 is not a program line number currently existing in the program, the next smaller program line number is used.
Listed below is a summary of DEL's reactions to various forms of the DEL command if the usual format is not used.

SYNTAX	EFFECT
DEL	?SYNTAX ERROR
DEL ,	?SYNTAX ERROR
DEL ,t	?SYNTAX ERROR
DEL -s[,t]	?SYNTAX ERROR

DEL Ø,t	deletes line zero, regardless of the value of t
DEL 1,-t	ignored, even if zero is the program's smallest line number
DEL s,-t	if s is greater than the program's smallest line number, then the result is ?SYNTAX ERROR, except when s is one and the program's smallest line number is zero
DEL s,-t	ignored if s is not zero and line number zero is the only program line
DEL s,-t	ignored if s is not zero and if s is less than or equal to the program's smallest line number
DEL s	?SYNTAX ERROR
DEL s [,]	ignored
DEL s,t	ignored if s is greater than t and if s is not zero

When DEL is used in the deferred-execution mode, it will operate as described above and will stop program execution. Typing CONT will not resume execution under these circumstances.

4.4 FRE

Execution mode: imm & def
Format: FRE (expr)

This command tells the user the amount of memory(in bytes) that is still available for use. Since MBASIC stores duplicate strings(such as A$=SANYOY and B$=SANYOY) only once, you may possibly end up with more memory than you had anticipated.

The result of FRE(expr) will be shown as a negative number if the number of free memory bytes is greater than 32767. If you do receive a negative number for FRE(expr) and would like to know the actual number of free bytes of memory, simply add 65536 to the negative number.

The number of bytes given by FRE(expr) are the ones occupying the space below the string storage and above the numeric array and string pointer array space (see memory map in Appendix N). HIMEM: can be set as high

as 65535. However, if it is set higher than the highest RAM memory location in your MPF-III, FRE will give you a number which does not make any sense and is greater than the memory capacity of the computer. (For more details on memory limits, refer to HIMEM: and POKE.)

Now assume that the contents of a string are changed during the execution of a program (for example, A$="BOAT" is changed to A$="SHIP"). In this case, MBASIC will not eliminate "BOAT", but will open a new file for "SHIP". Therefore, there are a lot of old characters which slowly fill down from HIMEM: to the top of the array space. Even though MBASIC will automatically perform the task of "cleaning up" when this old data runs into the free array space, the old characters may be battered if any of the free space is used for machine language programs or high-resolution page buffers.

If you use the statement

SCREEN

```
X=FRE(Ø)
```

in your program at regular intervals, MBASIC will "clean up" and keep such events from occurring.

4.5 HOME

Execution mode: imm & def
Format: HOME

The HOME command has no parameters. This command is used to move the cursor to the upper leftmost screen position within the scrolling window and to clear all existing text within the window. Using HOME is exactly the same as using CALL -936.

4.6 HTAB

Execution mode: imm & def
Format: HTAB aexpr

Suppose the line in which the cursor is situated has 255 positions, beginning from position 1 all the way to position 255. No matter how wide you have set the text window, positions 1 through 4Ø are on the current line, 41 through 8Ø are on the next line, etc. The use of HTAB enables you to move the cursor to a location on the screen that is \aexpr\ positions from the left edge of the current screen line.

HTAB's cursor movements are dependent on the left margin of the text window, but are independent of the line width. Although HTAB can move the cursor outside the text window, it can only be moved long enough to print one character. HTAB 1 can be used to put the cursor in the leftmost position of the current line and HTAB Ø puts the cursor at position 256. If \aexpr\ is negative or greater than 255, you will receive the error message

```
SCREEN
    ?ILLEGAL QUANTITY ERROR
```

NOTE: The structures of HTAB and VTAB are not parallel. HTABs beyond the right edge of the screen do not cause the

```
SCREEN
    ?ILLEGAL QUANTITY ERROR
```

message to occur, but makes the cursor jump to the next lower line and tab ((aexpr-1)MOD 4Ø)+1.

4.7 INVERSE AND NORMAL

```
Execution mode: imm & def
        Format: INVERSE
                NORMAL
```

These two commands are used to set video output modes. They have no parameters and do not influence the display of characters as you type them into the computer nor characters already on the screen.

INVERSE sets the video mode so that what was black on the screen is changed to white and what was white is changed to black. White characters originally on a black background is changed to black charactes on a white background.

NORMAL sets the video mode so that the usual white characters appear on a black background.

NOTE: INVERSE affects output and
      NORMAL affects both input and output.

4.8 LIST

```
Execution mode: def
        Format: LIST [linenum1][-linenum2]
                LIST [linenum2][,linenum2]
```

If both lineum1 and linenum2 have not been specified in a LIST instruction, with or without a delimiter, then the entire program is displayed on the screen. If linenum1 is specified without a delimiter, MBASIC displays the line numbered linenum1. If linenum1 and a delimiter are specified, then the program is displayed from the linenum1 through the end. If linenum1, a delimiter and linenum2 are present, then the program is listed from the line numbered linenum1 through the line numbered linenum2. If more than one line is to be listed but the line numbered linenum1 is not in the program, then the program will be listed from the next greater line number. If the line numbered linenum2 in the LIST statement does not exist in the program, then the next smaller line number that does appear in the program will be used.

The following commands can be used to list the entire program:

```
LIST Ø          LIST [,1-] Ø          LIST Ø [,1-] Ø
```

LIST linenum, Ø
will list the program starting from the line with line number linenum through the end of the program.

LIST, A
lists the entire program, then gives the error message

```
SCREEN

    ?SYNTAX ERROR
```

Before storing your program, MBASIC "tokenizes" your program lines, removes and inserts spaces whenever necessary. When LISTing, MBASIC lists the tokenized program lines. For instance,

```
100 B=+7/-8:A=-7
```

is changed to

```
100 B = + 7 / - 8:A = - 7
```

The LIST instruction can only be terminated by typing CONTROL C.

4.9 POS

Execution mode: imm & def
Format: POS (expr)

This command gives you the current horizontal position of the cursor on the screen, relative to the left hand margin of the text window. If the cursor is at the left margin, then 0 is returned. Although expr which is used in the POS command has no meaning and is only there to hold the parentheses apart, it must not be an illegal expr. A legal expr can be a number, a string or a variable name. If expr consists of a set of characters which may not be used as a variable name, the characters must be enclosed in quotation marks(").

NOTE: HTAB and TAB positions are numbered from 1, but POS and SPC positions are numbered from 0. Thus

```
PRINT TAB(20);POS(0)
```

causes 19 to be printed, and

```
PRINT SPC(20);POS(0)
```

causes 20 to be printed.

4.10 REM

Execution mode: imm & def
Format: REM {character|"}

The REM statement allows text, including blanks and statement separators, to be inserted in a program; even though whatever is typed after REM is ignored by the computer. REM is terminated by pressing the RETURN key ⟵┘.

When REM statements are listed, MBASIC places an extra space after REM, regardless of how many spaces were typed after REM by the user.

4.11 right arrow, left arrow, up-arrow and down-arrow

Execution mode: imm only
Format: right-arrow
left-arrow
up-arrow
down-arrow

The right-arrow key →, a key which is symbolized by an arrow pointing to the right on your keyboard, is used to move the cursor to the right. Each time the cursor moves, all the characters it crosses is stored in memory. This right-arrow key is especially useful when when only minor changes are required. Thus, it saves you from retyping an entire line of instruction if the line contains an error.

The left-arrow key ← is used to move the cursor to the left. As the cursor moves to the left, each character it crosses is erased from the program line which you are currently typing, no matter what the cursor is moving over.

If you now press the return key ⟵┘ in the program line which you are currently typing and press the left-arrow key ← afterward, then this line cannot be erased by the left-arrow key because it will cause the prompt character ( ]) to appear in column Ø of the next lower line, followed by the cursor. Therefore, the cursor frequently cannot be moved to column Ø of the screen by

using the left-arrow key <- for this reason.

The up-arrow | and down-arrow v keys are only used with electronic games and can be used to control upward or downward movements. However, these two keys cannot be used in program editing.

4.12 SPEED

Execution mode: imm & def
Format: SPEED=aexpr

The SPEED command regulates the speed at which characters are to be returned to the screen or sent to other input/output devices. The speed at which characters can be returned range from Ø through 255, where the slowest speed is Ø and the fastest speed is 255. Values which are out of this range will cause the computer to display the error message

SCREEN

```
?ILLEGAL QUANTITY ERROR
```

4.13 SPC

Execution mode: imm & def
Format: SPC (aexpr)

The SPC command must be used with a PRINT statement, and aexpr must be in parentheses. This command enables you to insert \aexpr\ spaces between the item previously printed and the next item to be printed. If SPC(Ø) is used, then no spaces are inserted.

\aexpr\ is required to be in the range from Ø through 255. Otherwise, you will receive the

```
SCREEN

    ?ILLEGAL QUANTITY ERROR
```

message. However, several SPC(aexpr) can be joined together to provide large spaces as shown in the following example.

```
PRINT SPC(375)  SPC(264)  SPC(380)
```

NOTE: Although HTAB is used to move the cursor to a absolute screen position (relative to the left margin of the text window), SPC(aexpr) is used to move the cursor a specified number of spaces away from the previously printed item. The location of this new position may be anywhere in the text window, depending on the location of the previously printed item.

Any spacing beyond the right edge of the text window causes spacing or printing to resume at the leftmost limit of the next lower line in the text window.

If \aexpr\ is a real, it must first be converted to an integer before it is used.

If the next non-space character of SPC is a left parenthesis, then it is interpreted as a reserved word.

4.14 TAB

Execution mode: imm & def
Format: TAB (aexpr)

Like SPC, the TAB command must also be used with a PRINT statement, and aexpr is required to be contained in parentheses. If the value of \aexpr\ is greater than the value of the current cursor position (relative to the left margin), TAB moves the cursor to the position that is \aexpr\ printing positions from the left margin of the text window. Otherwise, the cursor is not moved. TAB cannot be used to move the cursor to the left (use HTAB for this).

If TAB moves the cursor beyond the right edge of the text window, the cursor is moved to the left edge of the next lower line in the text window, and spacing

continues from there.

TAB(Ø) sets the cursor at position 256. If \aexpr\ is not in the range Ø through 255, then the message

```
SCREEN

    ?ILLEGAL QUANTITY ERROR
```

is displayed. TAB is interpreted as a reserved word if the next non-space character is a left parenthesis.

4.15 VTAB

Execution mode: imm & def
Format: VTAB aexpr

VTAB, which can be said to be an abbreviated form of "Vertical TAB", moves the cursor to the line that is located \aexpr\ lines down on the screen. The top line on the screen is line 1; the bottom line is line 24. The VTAB statement can be used to move the cursor vertically (either up or down), but never horizontally (to the right or left).

If aexpr is outside of the range 1 through 24, then the message

```
SCREEN

    ?ILLEGAL QUANTITY ERROR
```

will be displayed on the screen.

The cursor movement caused by VTAB is relative only to the top and bottom of the screen. VTAB ignores the text window. In the graphics mode, VTAB can be used to move the cursor into the graphics area of the screen. If VTAB moves the cursor to a line below the text window, all subsequent printing will occur on that line.

4.16 ESC A, ESC B, ESC C, ESC D

Execution mode: imm
(not allowed in defferred mode)

Format: ESC A
ESC B
ESC C
ESC D

The ESC key, when used in conjunction with the A, B, C, D keys, can be used to move the cursor. The functions of these keys are:

ESC A - Moves the cursor one position to the left
ESC B - Moves the cursor one position to the right
ESC C - Moves the cursor one line up
ESC D - Moves the cursor one line down

To enter ESC A, first press the ESC key to enter into the Screen Edit Mode and then enter the cusor movement keys - A, B, C, or D. Note that once a cursor movement key is entered, the MPF-III will exit the Screen Edit Mode.

Note that when using the cursor movement keys the characters over which the cursor is moved across are not affected, e.g., the ASCII codes of those characters are still stored in the RAM and screen buffer.

To modify a program line, you may use the LIST command to list that program line. Through the use of the ESC command, you may mqve the cursor to the first column of that program line. Remember that the next step is to type the SPACE BAR on your keyboard. Then you can copy the characters you don't want to change with the right handed arrow key -> and the REPT keys. As you proceed to the position where you intend to make corrections, you can type in the correct characters. In most cases, after you have typed in the correct characters you still have to use the -> and/or REPT keys to copy the characters you want to keep. Since each program line should be ended with a carriage return code, you have to type the ↵ key to end a program modification session.

When the ESC key is used together with the I, J, K, M keys, the cursor can be moved continuously without exiting the Screen Edit Mode.

The functions of ESC I, J, K, M are:

ESC K - Moves the cursor to the right
ESC J - Moves the cursor to the left
ESC I - Moves the cursor up
ESC M - Moves the cursor down

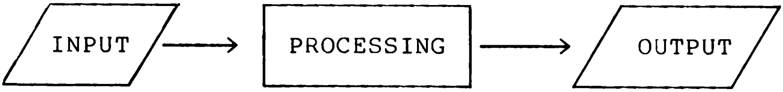
4.17 REPEAT

Execution mode: imm
(Not allowed in the deferred mode)
Format: REPT

The function of the REPT key is to type a character repeatedly. When the REPT key is typed at the same time with another key, that character will be typed repeatedly. If only the REPT key is typed, then the last entered character will be typed repeatedly.

CHAPTER 5
COMMANDS FOR ARRAYS AND STRINGS

In Chapter 3, we mentioned that the operations of a computer can be summarized as illustrated below:



If you can recall our discussion of the commands in Chapter 3, we only discussed commands dealing with the input and output operations of your MPF-III. Then, in Chapter 4, commands pertaining to editing and formatting were introduced. Since program editing is related to the input operations of a computer and formatting has to do with positioning your data into the desired format for output, it seems that we have not as of yet discussed commands concerning the processing or manipulation of data. Although the processing operation of a computer can be taken to mean to execute a program or a series of instructions, it can also be looked upon in terms of manipulating data before it is printed out as output. The commands for arrays and strings to store, recall or extract particular data items are: ASC, CHR$, DIM, LEFT$, LEN, MID$, RIGHT$, STORE and RECALL, STR$, and VAL.

5.1 ASC

```
Execution mode: imm & def
        Format: ASC (sexpr)
```

The ASC function displays a ASCII code for the first character of \sexpr\. ASCII codes in the range 96 through 255 will generate characters on MPF-III which reiterate those in the range Ø through 95. Although the results returned for CHR$(65) and CHR$(193) are both A, MBASIC does not consider the two to be the same character when using string logical operators.

If the argument of a string is Ø, it must be enclosed in quotation marks (") and quotation marks may not be included within the string. If it is a null string, the computer will display the error message

SCREEN

```
?ILLEGAL QUANTITY ERROR
```

If you try to use an ASC function on CONTROL @, you will receive the

SCREEN

```
?SYNTAX ERROR
```

message.

5.2 CHR$

```
Execution mode: imm & def
        Format: CHR$ (aexpr)
```

The CHR$ function displays the ASCII character which corresponds to the value of aexpr. If \aexpr\ is not in the range Ø through 255, then the

SCREEN

```
?ILLEGAL QUANTITY ERROR
```

message is displayed. In the event that \aexpr\ is a real, it will first be converted to an integer.

5.3 DIM

Execution mode: imm & def
Format: DIM var subscript[{var subscript}]

During an execution of a DIM statement, the computer saves space for the aray with the name var. In general, the amount of space allocated for an array are as follows:

2 bytes for storing an array variable name
2 bytes for the size of the array
1 byte for the number of dimensions, and
2 bytes for each dimension

However, the amount of space allocated for the elements in an array may vary according to the type of array as discussed below.

Subscripts range from Ø to \subscript\. The number of elements in an array with n dimensions is

(\subscript1\+1)*(\subscript2\+1)*...*(\subscriptn\+1)

For example:

```
DIM SHOW (3,4,5)
```

sets aside 4*5*6 elements (12Ø elements). Elements characteristic to this array are:

```
SHOW (3,4,2)
SHOW (Ø,Ø,2)
.
.
.
etc.
```

An array can only have up to 88 dimensions, even though

each dimension can only contain one element. For instance, if DIM A(Ø,Ø,Ø...Ø) has 89 zeroes, then the error message

SCREEN

```
?OUT OF MEMORY ERROR
```

is displayed. However, if DIM A(Ø,Ø,...Ø) had 88 zeroes, then no error is generated.

Customarily, however, the size of arrays are frequently restricted to the amount of memory space available. Array elements are stored in memory as follows:

ARRAY ELEMENT	STORAGE IN MEMORY
INTEGER ARRAY ELEMENT	occupies 2 bytes (16 bits) each
REAL ARRAY ELEMENT	occupies 5 bytes (4Ø bits) each
STRING ARRAY VARIABLES	use 3 bytes each (1 for length, 2 for a location pointer) and is stored as an integer array when the array is dimensioned

For string array variables, it also takes an additional one byte per character as the strings are stored by the program.

In the program, if a dimensioned array has the same name as a previously dimensioned array, the error message

SCREEN

```
?REDIM'D ARRAY ERROR
```

will be given.

The individual strings in a string array are not dimensioned, but will increase or decrease in size as

necessary. For instance, the statement

```
NAME$(7) = "BATTERA"
```

produces a string with a length of 7. The statement

```
NAME$(7) = " "
```

de-allocates the space assigned to the string NAME$(7). A string may have a maximum of 255 characters.

5.4 LEFT$

Execution mode: imm & def
Format: LEFT$ (sexpr, aexpr)

The LEFT$ function will extract and display the left-most \aexpr\ characters, contained in \sexpr\. For example,

SCREEN
```
PRINT LEFT$("MPF-III",3)
```

will print

SCREEN
```
MPF
```

If any part of this command is omitted, and \aexpr\<1 or \aexpr\>255, then the error message

SCREEN
```
?ILLEGAL QUANTITY ERROR
```

will appear. If \aexpr\ is a real, it will first be converted to an integer.

If \aexpr\ is greater than LEN(sexpr), only the characters which make up the string are displayed and any extra positions are disregarded.

If the "$" is left out from the command name, MBASIC will consider LEFT to be an arithmetic variable name and the error message

```
SCREEN
    ?TYPE MISMATCH ERROR
```

will appear.

5.5 LEN

Execution mode: imm & def
Format: LEN (sexpr)

The LEN function can be used to display how many characters are in a string, between Ø and 255. If the argument of a string is a concatenation of strings whose total length is longer than 255 characters, you will be given the error message

```
SCREEN
    ?STRING TOO LONG ERROR
```

on your computer screen.

5.6 MID$

Execution mode: imm & def
Format: MID$ (sexpr, aexpr1 [,aexpr2])

In the string MID$, if there are two arguments sexpr and aexpr1, then the substring starting at the \aexpr1\th character of \sexpr\ and going on through the last character of \sexpr\ will be displayed. Take a look at the following example:

```
SCREEN
    PRINT MID$("MPFSOFT",2)
    PFSOFT
```

The effect of the following expressions listed below are the same:

MID$(sexpr,aexpr) = RIGHT$(sexpr, LEN(sexpr)+1-\aexpr\)

If there are three arguments in the string MID$, then \aexpr2\ characters of \sexpr\, starting from the \aexpr1\th character, and proceeding to the right will be returned. For instance, typing

```
SCREEN
    PRINT MID$("MPFSOFT",2,4)
```

will return

```
SCREEN
    PFSO
```

If \aexpr1\ is greater than LEN(sexp), then MID$ returns a null string. If \aexpr1\ + \aexpr2\ exceeds LEN(sexpr), or is greater than 255 (the maximum length of any string), anything extra which exceeds the length of sexpr or 255 is ignored. If LEN(A$)=255, then the result of MID$(A$,255,255) is the last character in A$, otherwise it is a null string.

If the range of \aexpr1\ or \aexpr2\ is not between 1 and 255, then you will receive the

```
SCREEN
    ?ILLEGAL QUANTITY ERROR
```

message. If the "$" is not contained the command name, MBASIC will consider MID to be an arithmetic variable name and will display the

SCREEN

```
?TYPE MISMATCH ERROR
```

message.

5.7 RIGHT$

Execution mode: imm & def
Format: RIGHT$ (sexpr, aexpr)

RIGHT$ will reiterate the rightmost \aexpr\ characters of \sexpr\. Look at the following example:

SCREEN

```
PRINT RIGHT$("MPF-III" + "WARE",8)
PF-III WARE
```

This command cannot be executed if any part of it is left out. If \aexpr\ is greater than or equal to LEN(sexpr) then the entire string will be returned.

If \aexpr\ is less than 1 or \aexpr\ is greater than 255 then the

SCREEN

```
?ILLEGAL QUANTITY ERROR
```

message is displayed.

If the "$" is left out of the command name, MPF-III will consider RIGHT to be an arithmetic variable name and will display the

SCREEN

```
?TYPE MISMATCH ERROR
```

message.

The effect of the following two expressions are identical:

```
RIGHT$(sexpr,aexpr)  = MID$(sexpr,LEN(sexpr) + 1 -
              \aexpr\)
```

5.8 STORE AND RECALL

```
Execution mode: imm & def
        Format: STORE avar
                RECALL avar
```

These two commands store and recall arrays (which can only be used in the LOADA and SAVEA formats) from cassette tape. Since array names are not stored with their values, a different name than that used with the STORE command may be used to read back an array.

The dimensions of the array specified by the RECALL statement must be the same as the dimensions of the array designated by the STORE statement. In other words, if the STORE statement is used to store an array dimensioned by DIM A(3,3,3), then the RECALL statement can be used to recall it into an array by DIM B(3,3,3). If the dimensions of the array in the RECALL and STORE statements are not the same, then you may end up with scrambled numbers or extra zeroes in the RECALLed array or receive the following message

SCREEN

```
?OUT OF MEMORY ERROR
```

The above error message will usually be given only when the total number of elements reserved for the recalled array is inadequate to contain all the elements of the stored array. Take a look at the following examples and their results to understand how STORE and RECALL operate.

```
DIM X(3,3,3)
STORE X
```
stores 4*4*4 elements on the cassette tape.

```
DIM Y(3,5)
RECALL Y
```

will give you the

```
SCREEN

    ERR
```

message and scrambled numbers in array Y, however, program execution will not stop.

```
DIM Y(3,9)
RECALL Y
```

will cause the

```
SCREEN

    ?OUT OF MEMORY ERROR
```

message to appear on your screen and execution of the program will stop. In this example, array Y only has 4*10 elements which is not adequate enough to hold all the elements of array X.

If the dimensions of the recalled array and stored array are identical, then any of the dimensions of the recalled array may be larger than the corresponding dimension of the stored array. For instance, DIM A(3,3,3) designates an array of three dimensions, each having the size of 4. However, if the dimension you specified in the array is not the LAST dimension of the recalled array which is larger than the last dimensions of the stored array, then you will end up with scrambled numbers in the recalled array. Extra zeroes will obviously be stored in the excess elements of the recalled array. For example, if the following commands were used to store array X,

```
DIM X(3,3,3)
STORE X
```

then you will discover that

```
DIM Y(6,3,3)
RECALL Y
```

and

```
DIM Y(3,6,3)
RECALL Y
```

will fill array Y with mixed-up numbers from array X; you will also discover that

```
DIM Y(3,3,6)
RECALL Y
```

works just as well, filling array Y's extra elements with zeroes.

For STOREing and RECALLing arrays with equal number of dimensions, we have already examined the following two rules:

1. Only the last dimension of the recalled array is permitted to be larger than the last dimension of the stored array.
2. The total number of elements recalled must be at least equal to the number of elements stored.

If there is a recalled array which fulfills the requirements of the second rule above and previous dimensions specified are the same as the stored array, then you may recall an array with more dimensions than the stored array. Program execution will continue and the

SCREEN

```
ERR
```

message will appear on your screen. For example, if

```
DIM Y(3,3,3,3)
RECALL Y
```

was used, it will work even though there will be extra zeroes in the array and the

```
SCREEN
    ERR
```

message will be displayed. However, if

```
DIM Y(3,3,1,3)
RECALL Y
```

is used, array Y will be filled with scrambled numbers and the

```
SCREEN
    ERR
```

message will also be displayed. If

```
DIM Y(3,2,1,1)
RECALL Y
```

is used, the

```
SCREEN
    ?OUT OF MEMORY ERROR
```

message will appear on your screen because the 4*3*2*2 elements in array Y are fewer than the 4*4*4 elements stored in array X.

Note that only real and integer arrays can be stored on cassette tape. In order to store string arrays on cassette tape, they must first be converted to an integer array using the ASC function.

Although subscripts or dimensions are not used to refer to variables by the commands STORE and RECALL, only arrays can be stored and recalled. Therefore, the program

```
100 X(5) = 13
110 X = 16
120 STORE X
```

does NOT take the variable X (which is 16 in the example) to be stored on tape, but takes the array elements X(0) through X(10) and stores them on tape. In most cases, arrays are dimensioned to eleven elements by default.

When storing arrays, no prompting message or any other signal will be given. Before execution of the STORE instruction, make sure that your recorder is set in record mode. The beginning of a recording is indicated by the first "beep" and the end of a recording is signalled by the second "beep".

The following program

```
100  DIM Y(3,8)
110  Y=4
120  RECALL Y
```

reads the 36(4*9) array elements Y(0,0) through Y(4,9) from tape. The value of the variable Y (which is 4 in the example) is not changed and as previously mentioned earlier, no prompt message will appear on the screen. The beginning and end of a recording will be indicated by a "beep" from your recorder.

If an array name, used in either a STORE or a RECALL command, has not been previously dimensioned or used with a subscript, you will receive the

SCREEN

```
?OUT OF DATA ERROR
```

message. However, even if an array name has been predimensioned or used with a subscript, STORE or RECALL may not be successfully executed if particular attention is not paid to the mode in which an array variable has been defined. If either STORE or RECALL is used in the immediate-execution mode and makes reference to an array name defined in a deferred-execution program line, then it is necessary for the deferred-execution program line to have been executed before the STORE or RECALL command.

Interruption of a STORE or RECALL command can be accomplished by pressing the RESET key.

When STORE or RECALL(which are reserved words in MBASIC) are used as the first characters of any variable name, execution of the commands may occur before the

SCREEN

```
?SYNTAX ERROR
```

message is displayed. For instance, the statement

SCREEN

```
STOREROOM=8
```

will cause the error message

SCREEN

```
?OUT OF DATA ERROR
```

to appear on your screen, unless an array name beginning with the characters RO has been previously defined. In this situation, MPF-III will try to STORE the array which is indicated by the first beep for the beginning of the recording and the second beep for the end of the recording. MPF-III will then display the

SCREEN

```
?SYNTAX ERROR
```

message while it tries to interpret the rest of the statement, "=8". Press the RESET key to abort the beeps and the error message.

As another example, the statement

```
SCREEN
    RECALLMENT=123
```

will cause the error message

```
SCREEN
    ?OUT OF DATA ERROR
```

to be displayed on your screen, unless an array name beginning with the characters ME has been previously defined. In this case, MPF-III will wait until it receives an array from the cassette recorder. In order to regain control of the computer, press the RESET key.

5.9 STR$

Execution mode: imm & def
Format: STR$ (aexpr)

The STR$ function transforms \aexpr\ into a string which has an equivalent value. Therefore, aexpr is evaluated first before it is transformed into a string. The value of STR$(200 000 000 000) is 2E + 11.

If \aexpr\ exceeds the limits for reals, then you will receive the

```
SCREEN
    ?OVERFLOW ERROR
```

message.

5.10 VAL

Execution mode: imm & def
Format: VAL (sexpr)

The VAL command makes out a string to be a real or integer and returns the value of that number. If the first character of sexpr is neither a space nor a number, then the value of the string is Ø. Each character after the first character is examined in the same manner(spaces, decimal points, +, -, E and numeric characters are all acceptable) until the first non-numeric character is encountered. The first non-numeric character and all of the following characters are then disregarded by the computer and the string will be evaluated as a real or an integer.

If the length of sexpr exceeds 255 characters, then the error message

SCREEN

```
?STRING TOO LONG ERROR
```

will be displayed on your screen.

If the value of a string is greater than 1E38, or if the number consists of more than 38 digits which include trailing zeroes, the error message

SCREEN

```
?OVERFLOW ERROR
```

will be given.

CHAPTER 6
COMMANDS REGARDING FLOW OF CONTROL

The commands in this chapter will help you to regulate or control the sequence in which lines of instructions are to be executed and which instructions are to be conditionally reiterated or "looped" in your program. You will discover that these commands will be used most often to branch to subroutines (such as error-handling routines) in your program and then to return control to the main program routine. These types of commands provide you with the means to take care of anticipated errors which may occur during program execution without causing the system to "hang" or interrupting the execution of the program.

6.1 FOR...TO...STEP

Execution mode: imm & def
Format: FOR real avar = aexpr1 TO aexpr2 [STEP aexpr3]

The FOR...TO...STEP instruction can be used to repeatedly execute a loop, which contains a set of instructions, until \aexpr2\ reaches a certain value. Upon execution of FOR, avar is given the value of \aexpr1\. The statements following the FOR instruction are then executed until MPF-III encounters a

NEXT avar

where the avar in the NEXT statement is the same as the avar in the FOR statement. For more details on NEXT, refer to section 6.5 of this chapter. \avar\ is then incremented by the value of \aexpr3\ (or if \aexpr3\ is not given, avar is automatically incremented by 1). After avar has been incremented, it is then compared to the value of \aexpr2\ and program execution will proceed as follows:

* if avar is greater than \aexpr2\, then program execution will resume at the statement following NEXT,

* if avar is less than or equal to \aexpr2\, then execution will resume at the statement following FOR.

Note that although \aexpr1\, \aexpr2\ and \aexpr3\ of the FOR statement may be reals, real variables, integers, or integer variables, avar has to be a real variable. If avar is assigned an integer variable, the error message

SCREEN

```
?SYNTAX ERROR
```

will appear on your screen.

Since the comparison of avar to \aexpr2\ occurs after incrementing avar, the set of instructions contained in the FOR...NEXT loop are always executed at least once.

Each loop uses 16 bytes in memory.

In MBASIC, FOR loops are allowed to be nested 10 levels deep. If FOR loops have been nested more than 10 levels deep, the error message

SCREEN

```
?OUT OF MEMORY ERROR
```

will be presented. On the other hand, to avoid receiving the

SCREEN

```
?NEXT WITHOUT FOR ERROR
```

message, make sure each nested loop is entirely contained within the next outer loop. Also, MBASIC requires the FOR statement and the NEXT statement to be contained in the same line when the FOR...NEXT loop is to be executed in the immediate-execution mode.

NOTE: A space should not be placed between T and O if the letter A is used just before TO. The statement

```
FOR A = ALPHA TO 16
```

is legal, however, the statement

```
FOR A = ALPHA T O 16
```

will be interpreted as FOR A = ALPH AT 'O16 and will be given the

SCREEN

```
?SYNTAX ERROR
```

message when an attempt is made to execute it.

6.2 GOSUB

Execution mode: imm & def
Format: GOSUB linenum

This command causes the program to branch to the line given by linenum. Upon execution of a RETURN statement, the program will branch back to the instruction after GOSUB. The execution of a GOSUB statement causes the address of the following statement to be stored on top of a "stack" of these addresses to let the program know where to return to later on in the program. To remove the top address in the RETURN "stack" of addresses, use the RETURN or POP command.

If the line number given by linenum does not exist in the program, then you will receive the error message

SCREEN

```
?UNDEF'D STATEMENT ERROR IN XXXX
```

is displayed, where XXXX in the error message represents the program line of the GOSUB statement. However, if GOSUB is used in the immediate-execution mode, the error message will be shown as follows:

SCREEN

```
?UNDEF'D STATEMENT ERROR
```

and does not display the line number of the offending GOSUB.

SCREEN

```
?OUT OF MEMORY ERROR
```

will be given if you attempt to nest GOSUB's more than 25 levels deep.

Each GOSUB, which has not yet been RETURNed, takes up 6 bytes of memory.

6.3 GOTO

```
Execution mode: imm & def
        Format: GOTO linenum
```

The GOTO statement causes the program to branch to the line whose line number is indicated by linenum. If the linenum is not specified or if the linenum given does not exist, then the error message

```
SCREEN

    ?UNDEF'D STATEMENT ERROR IN XXXX
```

will be given, where XXXX represents the line number of the program line containing the GOTO statement.

6.4 IF...THEN and IF...GOTO

```
Execution mode:  imm & def
Format: IF expr THEN instruction [{;instruction }]
        IF expr THEN [GOTO] linenum
```

The IF statement (in any of the above formats) can be used to conditionally test a supposition and make the program execute all of the specified instructions in the THEN portion of the statement if the condition in the IF portion of the statement is true. Generally, the value of expr will determine whether or not the instructions contained in the THEN portion will be executed as described below:

* If the value of expr > < Ø, AND if its absolute value > 2.93873e-39, then the expr is evaluated to be true and the instructions in the THEN portion of the statement will be executed.

* If the value of expr = Ø OR if its absolute value < 2.93873E-39, then the expr is evaluated to be false and the instructions in the THEN portion of the statement are ignored. Program execution will then continue with the next program line number.

Sometimes the execution of the THEN portion is also determined by what execution mode is being used.

* If the IF statement being executed in the immediate-execution mode and if the value of expr is zero, then the rest of the program will be ignored by MBASIC.

In some cases, the contents of expr can also be a determinant factor in the execution of the IF statement.

* If expr contains string expressions and string logical operators, then the value of expr is determined by the comparison of the alphabetical ranking of the string expressions as defined by the ASCII codes for the characters in expr. (For more details on ASCII codes, see Appendix J.)

If the IF statement is given in the form

IF aexpr THEN

then no error message is given and the statement is considered to be legal.

If an IF does not have a corresponding THEN or if a THEN is placed in a program without a corresponding IF, you will receive the

SCREEN

```
?SYNTAX ERROR
```

message on your screen.

In MBASIC, the value of expr in an IF statement is not permitted to be a string expression, however, string variables and strings may be used under the following conditions:

(1) The value of expr is evaluated as non-zero if expr is a string expression of any kind regardless of the fact that expr is a string variable which has been given no value or a zero or a null string.

(2) The literal null string

IF " " THEN...

will be evaluated as zero.

(3) The statement

IF string THEN...

will produce the

```
SCREEN
?FORMULA TOO COMPLEX ERROR
```

message to appear if it is executed more than two or three times in a program.

(4) The value of expr is evaluated as zero if expr is a string variable and if the null string was given to any string variable by the previous statement. For example, when the following program

```
SCREEN
100 IF X$ THEN PRINT "X$"
110 IF Y$ THEN PRINT "Y$"
120 IF Z$ THEN PRINT "Z$"
```

is executed, it will print

```
SCREEN
X$
Y$
Z$
```

as output since the strings X$, Y$, and Z$ all evaluate as non-zero. Adding the following line

```
80 P$ = " "
```

to the program, however, will cause the three strings to be regarded as zero and no output will be given. If line 100 is deleted from the program and if a statement such as

```
90 Q=3
```

is inserted into the program, then all three strings will be evaluated as non-zero again.

(5) If the letter A is placed right before THEN as in the statement

```
SCREEN
IF CALCULA THEN 300
```

then it could be interpreted as

```
IF CALCUL AT HEN 300
```

and the

```
SCREEN
?SYNTAX ERROR
```

message will be displayed when it is executed.

In so far as the IF statement format is concerned, the three statements below are all considered to be the same:

```
IF A=8 THEN 200
IF A=8 GOTO 200
IF A=8 THEN GOTO 200
```

6.5 NEXT

Execution mode: imm & def
Format: NEXT [avar]
NEXT avar [{,avar}]

NEXT is used to terminate the FOR...NEXT loop started by a FOR statement. When NEXT is executed, program execution will either continue with the instruction following NEXT or return back to the corresponding FOR statement, depending on the parameters in the FOR statement. For more details on the FOR statement, refer to section 6.1 of this chapter.

If several avars are used in a NEXT statement, they must be listed in the proper order so that a FOR...NEXT loop is nested within another FOR...NEXT loop. If avars are not listed in the proper order, the

SCREEN

```
?NEXT WITHOUT FOR ERROR
```

message will appear on your screen.

If an avar is not specified in a NEXT statement, the program will resume execution with the FOR loop that is still in effect. If the program cannot find the active FOR loop for the corresponding NEXt statement, the error message

SCREEN

```
?NEXT WITHOUT FOR ERROR
```

will be displayed. NEXT with no avar specified executes faster than NEXT avar.

In general, the FOR statement and its associated NEXT statement NEXT statement must be executed in the same line while operating in the immediate-execution mode. If a NEXT statement in the immediate-execution mode is executed while a FOR statement in the deferred statement is still in effect, the NEXT statement will cause a jump to the deferred-execution program. A FOR and NEXT statement both executed in the immediate-execution mode and not on the same line will cause the message

SCREEN

```
?SYNTAX ERROR
```

to be presented unless there are no lines in between FOR and NEXT; and neither a FOR nor an avar is associated with NEXT as shown in the example below.

SCREEN

```
]FOR N = 1 TO 5 : PRINT N
 1
]NEXT
 2
]NEXT
 3
]NEXT N
?SYNTAX ERROR IN XXXX    (where XXXX represents
                          some line number)
```

6.6 ON...GOTO and ON...GOSUB

```
Execution mode: def only
        Format: ON aexpr GOTO  linenum {[,linenum]}
                ON aexpr GOSUB linenum {[,linenum]}
```

These two statements both branch to the line number given by the \aexpr\th item in the list of linenum in the statement after either GOTO or GOSUB. The only difference is that in ON...GOTO, GOTO is executed and in ON...GOSUB, GOSUB is executed.

If the value of \aexpr\ is Ø or greater than the number of listed alternate linenums, (but is less than 256), then the program ignores this instruction and the execution of the program will continue with the next statement.

If the value of \aexpr\ is not in the range of Ø through 255, then the

SCREEN

```
?ILLEGAL QUANTITY ERROR
```

message will be given.

6.7 ONERR GOTO

```
Execution mode: def only
        Format: ONERR GOTO linenum
```

The ONERR GOTO statement sets a flag which causes the program to branch to the given line number indicated by linenum when an error occurs. In order for ONERR GOTO to function properly in your MBASIC program, it must be executed before an error occurs so that it can be used to avoid printing error messages and to avoid stopping program execution. The command POKE 216,Ø can be used to reset the error-detection flag if you want to have the normal error messages displayed.

In MBASIC, each type of error which occurs in relation to the ONERR GOTO command is given a code number. The code of the error which occurred last is stored in memory location 222. To retrieve error codes in order to see what type of error was encountered, type

SCREEN

```
PRINT PEEK(222)
```

For a listing of ONERR GOTO error codes and error messages, refer to Appendix J of this manual.

Errors which occur inside FOR...NEXT loops or in a GOSUB...RETURN subroutine will disrupt the pointers and RETURN stacks. When such an error occurs, your error-handling routine must return to the FOR or GOSUB statement to restart the loop or subroutine. Returning to the NEXT or RETURN statement instead of the FOR or GOSUB statement will cause the

?NEXT WITHOUT FOR ERROR or

?RETURN WITHOUT GOSUB ERROR

message to be printed.

If there are two GET errors in a row and the error-handling routine in ONERR GOTO is used with RESUME, the program will "hang". To regain control of the computer, press RESET CONTROL C ⟵┘. If GOTO is used to end the error-handling routine instead of RESUME, then everything will function properly except that the 87th INPUT error causes a jump to the Monitor.

If the 43rd INPUT error is encountered when used in the TRACE mode or in a program which contains a PRINT statement, ONERR GOTO will jump to the Monitor.

However, it will function properly if RESUME is used. If the program "hangs", press RESET CONTROL C [⟵] to regain control of the computer and to return to MBASIC.

Any of the problems previously discussed in regards to ONERR GOTO can be avoided if your error-handling routine contains a CALL to the following assembly-language subroutine:

* if in MONITOR, input hexadecimal data:

  68 48 68 A6 DF 9A 48 98 48 60

* if in MBASIC, input decimal data:

  104 168 104 166 223 154 72 152 72 96

For instance, POKE can be used to store the decimal numbes into locations 768 through 777. After all the decimal numbers have been POKEd, then CALL 768 can be used in the error-handling routine. For more details on how the assembly-language subroutine works, see the Assembly-language fix in Appendix J.

6.8 POP

Execution mode: imm & def
Format: POP

The POP command has no parameters or options. Upon execution of this command, you will discover that POP has the same effect as a RETURN command except that it will not branch to the statement which immediately follows the last GOSUB statement executed. (For more information on RETURN, refer to section 6.10 of this chapter.) When the next RETURN command is encountered, the program will branch to one statement beyond the second most recently executed GOSUB instead of branching to one statement beyond the last GOSUB statement executed. Because this command "pops" one address off the top of the stack of RETURN addresses, the command is known as the "POP" command.

If the execution of POP occurs before the program encounters a GOSUB, then the

SCREEN

```
?RETURN WITHOUT GOSUB ERROR
```

message will appear on your screen since there are no return addresses on the stack.

6.9 RESUME

Execution mode: def only
Format: RESUME

When RESUME is encountered at the end of an error-handling routine, it will cause the program to continue execution at the beginning of the statement in which the error was encontered.

If the program comes across RESUME before an error occurs, the error message

SCREEN

```
?SYNTAX ERROR IN 65278
```

may be displayed or other peculiar happenings may occur. In most cases, however, the computer will stop execution of your program or cause the system to "hang".

If there is an error in an error-handling routine, RESUME may cause the program to be trapped in an infinite loop. In this case, press RESET CONTROL C to escape and to return to MBASIC.

If RESUME is used in the immediate-execution mode, it may cause the system to "hang" and cause the

SCREEN

```
?SYNTAX ERROR
```

message to be given or it may start execution of an existing or deleted program.

6.10 RETURN

Execution mode: imm & def
Format: RETURN

No parameters or options are associated with the RETURN command. Execution of RETURN causes the program to branch to the statement which immediately comes after the last GOSUB statement executed. The address of this statement to which the program branched is the top one on the RETURN "stack". (Refer to section 6.2 on GOSUB and section 6.8 on POP for more details.)

If RETURN statements are encountered in a program once more than GOSUB statements are encountered, then the

SCREEN

```
?RETURN WITHOUT GOSUB ERROR
```

message will be displayed on your screen.

CHAPTER 7
COMMANDS ASSOCIATED WITH GRAPHICS

Thus far, we have only discussed commands to input, process and output data which only consists of characters (alphabetic letters, numbers, spaces, etc.). Another characteristic of your MPF-III is its ability to draw graphic pictures according to programming instructions given by the user. The type of commands you will be using in those programming instructions are described in this chapter, you should take special note as to which commands may be used in one of both of the graphic modes listed below:

(1) Low-resolution graphics mode
(2) High-resolution graphics mode

For a brief preview of these graphic modes, please refer back to sections 1.3.6 and 1.3.7 of Chapter 1.

7.1 COLOR

Execution mode: imm & def
Format: COLOR [=] aexpr

The COLOR command is an MBASIC instruction which must first be used to set the value of color before plotting in the low resolution graphics mode. If the value of COLOR is not assigned, then the color black will be used for plotting. Hence, you may discover that you will not be able to plot figure.

COLOR is used to set the color in the low-resolution graphics mode. If the value \aexpr\ is a real, MPF-II will first convert it to an integer. The \aexpr\ must be in the range Ø through 255 and is interpreted as modulo 16. If the color code given by the user is greater than 15, it will repeat the colors shown below (that is; Ø, 16, 32, 48 are black; 1, 17, 33, 49 are green and so on). Color names and their corresponding color codes are listed below.

CODE	COLOR	CODE	COLOR	CODE	COLOR	CODE	COLOR
Ø	BLACK	4	GREEN	8	PURPLE	12	WHITE
1	GREEN	5	ORANGE	9	GREEN	13	ORANGE
2	PURPLE	6	BLUE	1Ø	PURPLE	14	BLUE
3	WHITE	7	WHITE	11	WHITE	15	WHITE

To set COLOR to Ø, use the GR command. Note that in the command format above, the use of the replacement sign "=" is optional when setting COLOR.

If you would like to know the COLOR of a specific point on the screen, use the SCRN command. For more details on SCRN, refer to section 7.9 of this chapter.

If COLOR is used in the high-resolution graphics mode, it has no effect whatsoever and is consequently ignored. If used in the TEXT mode, however, COLOR is a factor used to determine which character is to be set on the screen by the PLOT instruction. For further details on PLOT, see section 7.8 of this chapter.

7.2 GR

Execution mode: imm & def
Format: GR

The GR instruction is used to set the graphic mode to the low-resolution graphics mode for plotting.

The GR command converts the screen to the low-resolution graphics mode (40x40) and leaves four lines for text at the bottom of the screen. When this command is executed, the screen is cleared to black and the cursor is moved to the text window. The execution of GR also sets COLOR to 0.

If GR (which is a MBASIC reserved word) is used as the first characters of a variable name, then it is possible that the GR command will be executed prior to displaying the

```
SCREEN

    ?SYNTAX ERROR
```

message. Because of this, an attempt to execute

```
GRAB=9
```

will result in unintentionally clearing the screen to black.

7.3 HCOLOR

Execution mode: imm & def
Format: HCOLOR [=] aexpr

HCOLOR is used to set the color to be used for plotting in the high-resolution graphics mode.

This MBASIC statement is used to set the color for plotting in high-resolution graphics mode. The value of \aexpr\ must be in the range 0 through 7. Color names and their corresponding color codes are as follows:

CODE	COLOR	CODE	COLOR
Ø	BLACK1	4	BLACK2
1	GREEN(depends on TV)	5	ORANGE(depends on TV)
2	PURPLE(depends on TV)	6	BLUE(depends on TV)
3	WHITE1	7	WHITE2

Note that there are four color codes whose color depends on your TV control settings. Your TV will also determine the color of a high-resolution dot plotted with a command such as HCOLOR=3(white) as discussed below.

If the x-coordinate of the dot is an even number, then HCOLOR=3 will turn out to be the color blue.

If the x-coordinate of the dot is an odd number, then HCOLOR=3 is the color green.

If both (x,y) and (x+1,y) are to be plotted, then HCOLOR=3 is white.

HGR, HGR2 or RUN cannot be used to change HCOLOR. A high-resolution graphics plot executed before the first HCOLOR statement will turn out to be an indeterminate color.

7.4 HGR

Execution mode: imm & def
Format: HGR

HGR is used to set the high-resolution graphics mode. This command is used to change the screen to the high-resolution graphics mode (28Øx16Ø) and leaves only four lines visible for text at the bottom of the screen. When HGR is executed, it clears the screen to black and displays page 1 of the high-resolution screen in memory (8K - 16K). HGR does not affect HCOLOR or the text screen memory. Although the text window is at full screen when this command is executed, only the bottom four lines of text will be displayed on the screen. The cursor will only be visible on the screen if it is first moved to the four-line text area.

Use of the reserved word HGR as the first characters of a variable name may cause HGR to be executed before the

SCREEN

```
?SYNTAX ERROR
```

message is given on your screen. For example, if the instruction

```
HGRNO=1Ø
```

was inputted, then your program may be erased and you will find yourself unexpectedly in the high-resolution graphics mode.

7.5 HGR 2

Execution mode: imm & def
Format: HGR2

HGR2 is the second area of high-resolution graphics and will clear the screen to black.

This command is used to convert the screen to the full-screen, high-resolution graphics mode (28Øx192) and clears the screen to black. Upon execution of HGR2, the second page contained in high-resolution memory (16K-24K) will be displayed on your screen and the text screen memory will not be affected. If your system contains more than 24K of memory, you can use HGR2 instead of HGR in order to maximize the memory space available for storing programs.

Use of the reserved word HGR2 as the first characters of a variable name will cause HGR2 to be executed first before the

SCREEN

```
?SYNTAX ERROR
```

message is displayed. For example, the instruction

```
2ØØ IF A=1Ø THEN HGR2PARTS=3Ø
```

will suddenly fill the screen with blank spaces and may

have erased the program lines before this line was executed.

7.6 HLIN

Execution mode: imm & def
Format: HLIN aexpr1, aexpr2 AT aexpr3

HLIN is used to draw a horizontal line.

SCREEN
```
10 GR:COLOR=3
20 HLIN 1,20 AT 5
```

This program will cause a horizontal line to be drawn at vertical coordinate 5 starting from X coordinate 1 to 20.

The HLIN command can be used to draw a horizontal line from (\aexpr1\,\aexpr3\) to (\aexpr2\,\aexpr3\) on the screen in the low-resolution graphics mode. The color used to draw the line will be determined by the color used by the last executed COLOR statement. (If there was no previously specified color, then COLOR=0 wll be used.)

The arithmetic expressions \aexpr1\ and \aexpr2\ must be in the range 0 through 39 and \aexpr3\ must be in the range 0 through 47, otherwise, the

SCREEN
```
?ILLEGAL QUANTITY ERROR
```

message will be presented on the screen.

If this command is used in the high-resolution graphics mode, it will have no effect.

NOTE: The "H" character in this command stands for "horizontal" and not "high-resolution". With the exceptions of HLIN and HTAB commands, other commands starting with the letter "H" stands for high-resolution.

7.7 HPLOT

Execution mode: imm & def
Format: HPLOT aexpr1, aexpr2
HPLOT TO aexpr3, aexpr4
HPLOT aexpr1, aexpr2 TO aexpr3, aexpr4 [{TO aexpr,aexpr}]

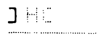
HPLOT is a instruction used in the high resolution graphics mode to plot horizontal and vertical lines.

SCREEN

```
10 HGR:HCOLOR=3
20 HPLOT 20,20
30 HPLOT 0,0 TO 39,0
40 HPLOT TO 20,40
```

The above program will plot the following figure.

SCREEN



Let's take a look at the HPLOT instructions in this program.

1) HPLOT 20,20 — Plots a dot at X=20, y=20

2) HPLOT 0,0 TO 39,0 — Plots a line starting from the origin (0,0) to (39,0)

3) HPLOT TO 20,40 — Plots a line starting from the last dot plotted in the last instruction to 20,40

If this instruction HPLOT aexpr1, aexpr2 is given, then MPF-III will plot a high-resolution dot at (\aexpr1\,\aexpr2\) using the same color used in the last executed HCOLOR statement. (If the value of HCOLOR was not previously specified, then an indeterminate color will be used.)

If the instruction HPLOT TO aexpr3, aexpr4 is used, then MPF-III will draw a line starting from the most recently plotted dot to the dot with the location specified by (\aexpr3\,\aexpr4\). If there was no previously plotted dot, then a line will not be drawn. The color used to draw the line will be the color used by the last HCOLOR statement executed.

If the instruction HPLOT aexpr1, aexpr2 TO aexpr3, aexpr4 [{TO aexpr,aexpr}] is given, then MPF-III will plot a line from (\aexpr1\,\aexpr2\) to (\aexpr3\,\aexpr4\) using the color designated by the last HCOLOR statement executed. In this case, the line may be extended indefinitely (provided that it is still within the limits of the screen and does not exceed 239 characters per line). This instruction can be extended by appending

TO aexpr5, aexpr6 TO aexpr7, aexpr8 TO aexpr9, aexpr10

and so on to the command. For example,

```
HPLOT 0,0 TO 100,0 TO 100,200 TO 0,200 TO 0,0
```

will plot a rectangle on your high-resolution screen.

Either HGR or HGR2 must come before a HPLOT instructin to protect your data, program and variables from being destroyed.

If \aexpr1\ and \aexpr3\ are not in the range 0 through 279; and \aexpr2\ and \aexpr4\ are not in the range 0 through 191, then the

SCREEN

```
?ILLEGAL QUANTITY ERROR
```

message will be shown if you try to plot a dot.

7.8 PLOT

Execution mode: imm & def
Format: PLOT aexpr1, aexpr2

PLOT instruction is used to PLOT dots in the low

resolution graphics mode.

When the following program is executed, a point will be plotted at the location whose horizontal coordinate is 1Ø and vertical coordinate is 2Ø -- using the color purple.

```
10 GR:COLOR=3
20 PLOT 10,20
```

Examine the following program.

SCREEN

```
10 GR:COLOR=2
20 FOR T=1 TO 20
30 PLOT 1,T
40 NEXT
```

This program will cause a vertical line to be drawn at horizontal coordinate 1 and vertical coordinate 1 - 2Ø.

The PLOT command is used to plot a dot specified by the location (\aexpr1\,\aexpr2\) in the low-resolution graphics mode using the color designated by the last COLOR statement executed. (If no color is specified, then COLOR=Ø is used.)

The arithmetic expression \aexpr1\ must be in the range Ø through 39 and \aexpr2\ must be in the range Ø through 47. If \aexpr1\ and \aexpr2\ are not within their respective ranges, then the

SCREEN

```
?ILLEGAL QUANTITY ERROR
```

message is displayed.

If PLOT is executed in the TEXT mode or in a combination of GRaphics-plus-text mode when \aexpr2\ is in the range 4Ø through 47, then MPF-III will print a character where the dot would have been plotted. (one character takes up the equivalent space of two low-resolutin graphics dots stacked vertically.)
In the graphics mode, the origin is at (Ø,Ø) which is the uppermost left corner of the screen.

7.9 SCRN

Execution mode: imm & def
Format: SCRN (aexpr1, aexpr2)

SCRN can be used to display the color code of a particular plotted dot.

```
10 GR:COLOR=3
20 VLIN 1,20 AT 10
30 PRINT SCRN (10,10)
```

When this program is executed, a vertical line will be drawn at the location whose horizontal coordinate is 10 starting at vertical coordinate 1 through 20. Then, in line 30, the instruction will cause the color code of the color used to plot the dot at (10,10) to be printed.

When SCRN is used in the low-resolution graphics mode, it will cause the color code of the point plotted at the location (\aexpr1\,\aexpr2\) to be returned on the screen. In the low-resolution graphics mode, \aexpr1\ (the x-coordinate) must be in the range 0 through 39 and \aexpr2\ (the y-coordinate) must be in the range 0 through 47. In spite of the fact that points can be plotted at a designated screen position (x,y) which must be in the respective limits discussed above, the SCRN command will accept the values in the range 0 through 47 for both x and y.

Use of the SCRN command with an aexpr1 which is NOT in the range 0 through 39 will cause MPF-III to return the color code for the point whose x-coordinate is (\aexpr1\-40) and whose y-coordinate is (\aexpr2\+16). If you are in the GRaphics-plus-text mode and the y-coordinate (\aexpr2\+16) is in the range 39 through 47, then SCRN will return the number associated with the text character at that position in the text area below the graphics portion of the screen. If (\aexpr2\+16) is between the numbers 48 and 63, then a number totally unassociated with anything on the screen will be returned.

When operating in the TEXT mode, SCRN will return numbers in the range 0 through 15 whose value is stored in the

(a) upper four bits, if aexpr2 is an odd number;
or

(b) lower four bits, if aexpr2 is an even number

of the character at the character position specified by (aexpr1+1,INT((aexpr2+1/2). Therefore, the following expression

```
CHR$(SCRN(x-1.2*(y-1))+16*SCRN(x-1,2*(y-1)+1))
```

will produce the character at (x,y).

When operating in high-resolution graphics mode, the SCRN command continues to "scan" the low-resolution graphics area. The number returned by the SCRN command is in no way associated with the high-resolution graphics display.

SCRN is interpreted as a reserved word only if a left parenthesis is the next non-space character.

7.10 TEXT

Execution mode: imm & def
Format: TEXT

TEXT is used to return to the text mode from the low or high resolution graphics mode.

The TEXT command moves the prompt and cursor to the last line of the screen and returns your screen display to the normal full-screen text mode (40 characters per line, 24 lines) from any of the graphics modes in MBASIC:

```
*GR:   low-resolution graphics mode  (40x40)
*HGR:  high-resolution graphics mode (280x160)
*HGR2: full-screen high-resolution graphics mode
                                      (280x192)
```

If the reserved word TEXT is used as the first characters of a variable name, then an error message is displayed. For example, if the statement

SCREEN

```
100 TEXTFILE = 128
```

is executed, the error message

SCREEN

```
?SYNTAX ERROR IN 100
```

will appear on your screen, where 100 represents the line number of the illegal variable name.

If the text window is not set to full-screen, the TEXT command can be used to set it to full screen.

7.11 VLIN

Execution mode: imm & def
Format: VLIN aexpr1, aexpr2 AT aexpr3

VLIN is used to draw a vertical line.

```
10 GR:COLOR=3
20 VLIN 1,20 AT 10
```

This program will plot a vertical line at horizontal coordinate 10 starting from vertical coordinate 1 to 20.

The VLIN command is used to draw a vertical line on the screen in the low-resolution graphics mode. When executed, VLIN will draw a vertical line from the given x-coordinate \aexpr1\ to x-coordinate \aexpr2\ at the y-coordinate \aexpr3\. The color used to draw the line is determined by the COLOR statement last executed. If no COLOR statement was executed prior to the execution of VLIN, then COLOR is automatically set to 0 for plotting.

Both \aexpr1\ and \aexpr2\ are required to be in the range 0 through 47. The \aexpr3\ must be in the range 0 through 39. If \aexpr1\, \aexpr2\ and \aexpr3\ are out of the specified ranges, then the

```
SCREEN

?ILLEGAL QUANTITY ERROR
```

message is presented on the screen.

If the screen is in text mode or in a combination of GRaphics-plus-text with \aexpr2\ in the range 40 through 47, the line wll be displayed as a line of characters instead of graphic dots.

7.12 PDL

Execution mode: imm & def
Format: PDL (aexpr)

The function PDL returns the current value of the paddle (game control) specified by \aexpr\. \aexpr\ must be in the range from Ø through 3, and the current value of the function is in the range from Ø through 255.

If the value of \aexpr\ is in the range from 4 through 255, then the PDL function will return an unexpected value in the range Ø through 255. This may bring about some undesirable side effects which may interfere with the execution of the program.

If your program contains two consecutive PDL instructions, it is suggested that some statements or a delay loop be inserted into the two PDL instructions so that the game control readings may be taken accurately. A delay loop such as the following

```
FOR I = 1 TO 1ØØ : NEXT I
```

may be entered.

If the value of \aexpr\ is negative or exceeds 255, then the message

```
?ILLEGAL QUANTITY ERROR
```

will be output.

If the value of \aexpr\ is in the range from 236 to 239, then PDL(aexpr) will cause

```
POKE -1654Ø + aexpr, Ø
```

so that PDL(236) may set the MPF-III to graphics mode, and PDL(237) sets text mode.

CHAPTER 8

COMMANDS USED TO CREATE HIGH-RESOLUTION SHAPES

In the previous chapter, you were shown primarily how to plot dots and lines. This chapter will introduce you to commands which will enable you to draw shapes in the high-resolution graphics mode. Step-by-step instructions on how to form, save, and use a shape table are presented in the first section of this chapter and are immediately followed by the commands used to draw high-resolution shapes: DRAW, ROT, SCALE, SHLOAD, and XDRAW.

8.1 How to Form and Use a Shape Table

In order to create and manipulate shapes in high-resolution graphics, MBASIC has incorporated the five following commands: DRAW, XDRAW, ROT, SCALE and SHLOAD. However, before you can use these commands, you will have to first describe the entire shape by using a "shape definition"and then be able to instruct the computer to draw it. A shape definition is composed of a sequence of plotting vectors which are stored in a series of bytes in MPF-III's memory. By using one or more of these shape definitions with their index, you will be able to create a "shape table" (which contains the coded characteristics of the figure to be drawn) from the keyboard and save it on disk or cassette tape for future use.

In a shape definition, each byte is broken up into three sections. In each section, the user will be able to specify a "plotting vector" to instruct the computer whether or not to plot a point and also give directions for vertical or horizontal movement. The DRAW and XDRAW commands step through each byte in the shape definition section by section, starting from the first byte and ending with the last byte of the definition. The shape definition is done when the byte containing all zeroes is encountered.

The arrangement of the three sections A, B, and C within one of the bytes in a shape definition is as follows:

SECTION:	C		B			A		
BIT NUMBER:	7	6	5	4	3	2	1	Ø
BIT FIELD :	D	D	P	D	D	P	D	D

What is D & P

CODES FOR PLOTTING IN A SHAPE TABLE

Plotting Vector	Codes	COMMENTS
↑	ØØØ	
→	ØØ1 or Ø1	Vectors for
↓	Ø1Ø or 1Ø	directional movement only
←	Ø11 or 11	
•↑	1ØØ	
•→	1Ø1	Vectors for plotting and
•↓	11Ø	directional movement
←•	111	

In the plotting section, we will describe how to form and use a shape table in order to draw the following figure

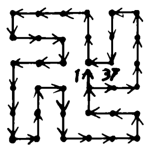


Figure 8-1

and how to rotate and enlarge it. The step-by-step instructions are as follows:

1. First, draw the desired shape and then draw the shape using short arrows as shown in Fig. 8-1. This figure uses 37 arrows or plotting vectors. Now, list out the arrows starting from arrow 1 and ending with arrow 37 as shown in Fig. 8-2.

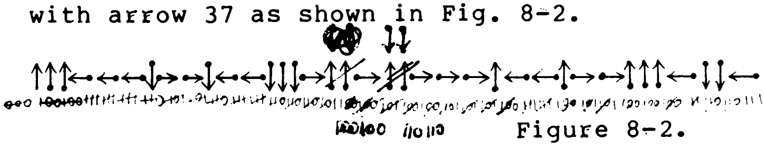


Figure 8-2.

2. Using the rules for plotting vectors in a shape table, find the code corresponding to each vector. (Note: arrow 1 simply represents a directional movement upward and is not to be plotted.) Once you have found the code for each vector, list the code out in the eight-bit format as specified by the organization of bits in a shape definition and fill out a table with sections A,B,C as shown in Fig. 8-3. After filling in the 6 bits in Sections A and B, the remaining 2 bits in Section C must be filled in with 00 since it is the code which is used to specify not to plot a dot. Using this method, fill in the code for the rest of the vectors until all the codes for all the vectors have been filled in. Finally, take the 8 bits of each of the vectors and convert it to a 16-bit value (or hexadecimal) by dividing the bits into groups of 4 and using the following table.(Table 8-1).

Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Table 8-1

C	B	A
ØØ	1ØØ	ØØØ
ØØ	111	1ØØ
ØØ	111	111
ØØ	1Ø1	11Ø
ØØ	11Ø	1Ø1
ØØ	111	111
ØØ	11Ø	11Ø
ØØ	1Ø1	11Ø
ØØ	~~1ØØ~~	~~1ØØ~~
ØØ	~~11Ø~~	1Ø1
ØØ	1Ø1	~~11Ø~~
ØØ	1Ø1	1Ø1
ØØ	111	1ØØ
ØØ	1ØØ	111
ØØ	1Ø1	1Ø1
ØØ	1ØØ	1ØØ
ØØ	111	1ØØ
ØØ	11Ø	11Ø
ØØ	ØØØ	111

Figure 8-3

Figure 8-4		
2Ø	Ø1	
3C	ØØ	
3F	Ø4	
2E	ØØ	
35		
3F		
36		
2E		
~~24~~ 24		
35		
~~2E~~		
2D		
3C		
27		
2D		
24		
3C		
36		
Ø7		ØØ

Figure 8-4

Ø1
ØØ
Ø4
ØØ
2Ø
3C
3F
2E
35
3F
36
2E
~~24~~
~~35~~
~~2E~~
2D
3C
27
2D
24
3C
36
Ø7
ØØ

Figure 8-5

3. Shape tables may contain up to 255 shape definitions. In constructing your shape table, you must first specify the number of shape definitions. In this case, we only have one shape definition, therefore, fill in Ø1 and ØØ as shown in the table Fig. 8-5. Then continue to fill in the table by determining what the distance of the first byte of the first shape difinition is from the starting position of the shape table. In this example, our shape definition is four bytes away from the start of our shape table. (Therefore, fill in Ø4 and ØØ in the table as shown in Figure 8-5.) Now fill in the hexadecimal codes in Figure 4. When the table has been completely filled in, place the byte ØØ at the end of the table to represent that this is the end of the shape definition. Your completed table should look exactly like the table in Figure 8-5.

4. Save the shape definitions in the RAM area which is not used either by the screen buffer or by the program. In this example, the shape definition is stored in the RAM area beginning from $1BØØ.

5. The memory location $E8 and $E9 is used as a pointer, pointing to the shape definition. The starting address of the shape definition - ØØ1B - is stored in locations $E8 and $E9.

6. Key in the BASIC program

SCREEN

```
10  GOSUB 150
20  HGR : HCOLOR=5
30  INPUT "A=";A : INPUT "X=";X : INPUT "Y=";Y
40  SCALE=C
50  ROT=B
60  FOR C=4 TO 10
70  FOR B=0 TO 20
80  DRAW 1 AT X,Y
90  FOR T=0 TO A : NEXT
100 XDRAW 1 AT X,Y
110 NEXT B
120 NEXT C
140 GOTO 10
150 POKE 232,0 : POKE 233,27
160 FOR I=0 TO 23 :  READ S : POKE 6912+I,S : NEXT I
170 RETURN
180 DATA 1,0,4,0,32,60,63,46,53,63,54,46,36,53,46,45,
    60,39,45,36,60,54,7,0
                                              36 53 46
```

This program will draw the shape which was discussed earlier and can enlarge, reduce, or rotate the shape.

SCALE is an instruction which can be used to specify the size of the shape to be drawn. (Refer to the section in this chapter on SCALE.)

ROT is an instruction to set the angle the shape is to be rotated. For more details, refer to the section in this chapter on ROT.

Shape definitions of your shape table may be stored permanently on tape. To store your shape table on tape, you must first save the length of the shape table

in locations $ØØ and $Ø1; and then use the following monitor commands to save the shape table on tape.

```
@Ø:18 ØØ (24 bytes)
@Ø.1W 1BØØ.1B16W
```

The SHLOAD instruction is used to load the shape table from the tape into the MPF-III. When loading the shape table, the MPF-III will beep twice. The first beep represents that the MPF-III has read in the starting and ending address of the shape table, namely, the contents of $ØØ and $Ø1. The second beep signifies that the MPF-III has read in the shape table.

8.2 DRAW

Execution mode: imm & def
Format: DRAW aexpr1 [AT aexpr2, aexpr3]

The DRAW command is used to draw high-resolution graphics shapes starting at the point given by \aexpr2\ as the x-coordinate and \aexpr3\ as the y-coordinate. Before the DRAW command can be properly executed, the shape definitions in the shape table must have been already loaded using the SHLOAD command. Note that the specifications of the color, rotation and scale of the shape must also have been given prior to the execution of the DRAW command. The shape drawn by this command is specified by the number of the shape definition given by \aexpr1\, which must be in the range 0 through 255 (number of shape definitions given in byte 0 of the shape table).

If the parameter \aexpr2\ is used, it must be in the range 0 through 278. The range of \aexpr3\ is from 0 through 191. If any of these parameters are out of their respective ranges, the error message

SCREEN

```
?ILLEGAL QUANTITY ERROR
```

will be presented on the screen.

If \aexpr2\ and \aexpr3\ are not specified in a DRAW statement, then the shape will be drawn starting at the last point plotted by the DRAW, XDRAW or HPLOT command last executed.

If DRAW is executed prior to loading a shape table in memory, the system will "hang" until you press RESET CONTROL C ⏎ to regain control of the computer. Attempts to execute DRAW before loading the shape table should be avoided since this may cause random shapes to be drawn all over the storage area for high-resolution graphics mode in memory. Although you may not be in the graphics mode, this could also destroy your program.

8.3 ROT

Execution mode: imm & def
Format: ROT=aexpr

This statement sets the orientation of high-resolution graphic shapes which are drawn by DRAW or XDRAW. The angular rotation used to draw a shape is determined by the value of \aexpr\, which must be a value in the range Ø through 255.

If ROT=Ø is executed, the shape will be drawn in the orientation with which it was defined. Generally speaking, for each increment of 16 in the value of \aexpr\, the shape is rotated 9Ø degrees clockwise. Therefore, ROT=16 will draw a shape with an angular rotation of 9Ø degrees (clockwise); ROT=32 will draw a shape rotated 18Ø degrees clockwise (so the shape is drawn upside down); ROT=48 will draw a shape rotated 27Ø degrees clockwise; and so on. ROT=64 has the same effect as ROT=Ø and will draw the shape in its original orientation.

Recognition of rotation values for ROT depends on the value of SCALE. (For more details on SCALE, see section 8.4 of this chapter.) If SCALE is set at 1, ROT only has four recognized values which are Ø, 16, 32, and 48. If SCALE=2, ROT only has eight recognized values; if SCALE=3, ROT has 16 recognized values and so on.

Unrecognized values for ROT will draw the shape with the orientation of the next lower recognized value. ROT is not interpreted as a reserved word unless the replacement sign "=" is the first non-space character immediately after "ROT".

8.4 SCALE

Execution mode: imm & def
Format: SCALE=aexpr

SCALE is used to set the size of high-resolution graphic shapes drawn by DRAW or XDRAW. When SCALE is executed, the integer value of \aexpr\ is multiplied by the size of the shape table. Note that the value of \aexpr\ must be in the range Ø through 255.

If SCALE=1, then the shape is drawn exactly as originally specified by the shape definition. If SCALE=2, then the shape is drawn twice the size of the shape table(where each vector is extended twice its size) and so on. However, if SCALE =Ø, then the shape will be drawn 255 times the size of the original shape; the maximum scale size with which the shape can be reproduced.

SCALE is not interpreted as a reserved word unless the replacement sign "=" is the first non-space character immediately after "SCALE".

8.5 SHLOAD

Execution mode: imm & def
Format: SHLOAD

This command loads a high-resolution graphics shape table from cassette tape. When a shape table is to be loaded in memory, it is loaded immediately below HIMEM: and HIMEM: is set immediately below the shape table to protect it. After the shape table is loaded, the starting address assigned to the shape table is then automatically given to MBASIC's shape routines. If a second shape table is loaded and replaces the first table already stored in memory, HIMEM: should be reset before the second table is loaded in order to save storage space in memory. To prepare shape tables on cassette tape, follow the instructions given in section 8.1 of this chapter.

On 16K systems, the HGR command is used to clear the top 8K of memory, starting from location 8192 to location 16383. Therefore, it may be necessary to set HIMEM: to 8192 prior to executing SHLOAD in order to force SHLOAD to place the shape table below page 1 of high-resolution graphics.

On 24K systems, HGR2 is used to clear memory from location 16384 to location 24575. Hence, it may be necessary to set HIMEM: to 16384 before executing SHLOAD. HGR and HGR2 should NOT be used if there is enough safe memory above location 24575 to hold your shape table.

SHLOAD can only be interrupted by RESET. If the characters of the reserved word SHLOAD are used as the first characters of a variable name, SHLOAD may be executed before you receive the

SCREEN

```
?SYNTAX ERROR
```

message. Execution of the statement

```
SHLOADER = 100
```

will cause the system to "hang" and at the same time, cause MBASIC to wait until it receives a program from the cassette recorder. To regain control of the computer, press RESET CONTROL C [⟵].

8.6 XDRAW

Execution mode: imm & def
Format: XDRAW aepxrl [AT aexpr2, aexpr3]

This command can be used to draw a high-resolution graphics shape on the screen and is basically the same as the DRAW command with the following exceptions:

* when XDRAW is first used to draw a shape and then used a second time with the same parameters, it can be used to easily erase the shape without erasing anything in the background;

* when XDRAW is used to draw a shape, it uses the complement of the color used to plot the existing points on the screen. The following pairs are complementary colors:

  BLACK and WHITE
  BLUE and GREEN

Like DRAW, the scale and rotation of the shape to be drawn must have been set by the SCALE and ROT commands before XDRAW can be executed. Similarly, \aexprl\ must be in the range Ø through 255. For more details on the DRAW command, refer to section 8.4 of this chapter.

CHAPTER 9
USEFUL
MATH FUNCTIONS

In this chapter, we have provided a list of some math functions which you may find helpful in computing trigonometric expressions, logarithms, absolute values, square roots, and exponentiations of the natural log of e. In addition to these functions, you will discover that there is a special function which may be used to generate random numbers. For simplicity, this chapter has been divided into two sections: (1) the intrinsic functions and (2) the derived functions of MBASIC.

9.1 The Intrinsic Functions: SIN, COS, TAN, ATN, INT, RND, SGN, ABS, SQR, EXP, LOG

All of MBASIC's intrinsic functions may be utilized in either the immediate or deferred execution mode. A short summary of some of MBASIC's arithmetic functions are listed below.

FUNCTION	VALUE OF FUNCTION
SIN (aexpr)	displays the sine of \aexpr\ radians
COS (aexpr)	displays the cosine of \aexpr\ radians
TAN (aexpr)	displays the tangent of \aexpr\ radians
ATN (aexpr)	displays the arctangent of \aexpr\ in radians
INT (aexpr)	displays the largest integer less than or equal to \aexpr\
RND (aexpr)	displays a random real number greater than or equal to Ø and less than 1 *
SGN (aexpr)	has a value of -1 if \aexpr\ < Ø, is Ø if \aexpr\ = Ø, and is 1 if \aexpr\ > Ø.
ABS (aexpr)	displays the absolute value of \aexpr\; that is, if \aexpr\ > = Ø, then the value of ABS (aexpr) is \aexpr\; otherwise, ABS (aexpr) is -\aexpr\.
SQR (aexpr)	displays the positive square root of \aexpr\ and will execute more quickly than ^.
EXP (aexpr)	displays the result of the exponentiation of e (where e is the base and \aexpr\ is the indicated power), e=2.718289.
LOG (aexpr)	displays the natural logarithm of \aexpr\.

* MBASIC generates random numbers according to the value that is given to \aexpr\.

If \aexpr\ < Ø, RND (aexpr) generates the same random number each time it is executed with the same \aexpr\. However, if a negative argument is used in the generation of a random number, then succeeding random numbers generated by positive arguments will follow the same sequence each time. Different negative arguments will generate a different random sequence. The principal reason for using a negative argument for RND is to initialize (or "seed") a repeatable sequence of random numbers, which can be very helpful in debugging programs that used RND.

If \aexpr\ = Ø, RND (aexpr) brings back the most recent random number generated (CLEAR and NEW commands have no effect on this.) This method of returning the last generated random number is sometimes easier than transferring the last random number to a variable for the purpose of saving it.

9.2 Derived Functions

Although not intrinsic to MBASIC, the functions listed below can be computed using the existing BASIC functions and can be easily put into effect by using the DEF FN function.

SECANT:
```
     SEC(X) = 1/COS(X)
```

COSECANT:
```
     CSC(X) = 1/SIN(X)
```

COTANGENT:
```
     COT(X) = 1/TAN(X)
```

INVERSE SINE:
```
     ARCSIN(X) = ATN(X/SQR(-X*X+1))
```

```
INVERSE COSINE:
     ARCCOS(X) = -ATN(X/SQR(-X*X+1))+1.5708



INVERSE SECANT:
     ARCSEC(X) = ATN(SQR(X*X-1))+(SGN(X)-1)*1.5708

INVERSE COSECANT:
     ARCCSC(X)  =  ATN(1/SQR(X*X-1))+(SGN(X)-1)*1.5708

INVERSE COTANGENT:
     ARCCOT(X) = -ATN(X)+1.5708

HYPERBOLIC SINE:
     SINH(X) = (EXP(X)-(EXP(-X))/2

HYPERBOLIC COSINE:
     COSH(X) = (EXP(X)+EXP(-X))/2

HYPERBOLIC TANGENT:
     TANH(X) = -EXP(-X)/(EXP(X)+EXP(-X))*2+1

HYPERBOLIC SECANT:
     SECH(X) = 2/(EXP(X)+EXP(-X))

HYPERBOLIC COSECANT:
     CSCH(X) = 2/(EXP(X)-EXP(-X))

HYPERBOLIC COTANGENT:
     COTH(X) = EXP(-X)/EXP(X)-EXP(-X)*2+1


INVERSE HYPERBOLIC SINE:
     ARGSINH(X) = LOG(X+SQR(X*X+1))

INVERSE HYPERBOLIC COSINE:
     ARGCOSH(X) = LOG(X+SQR(X*X-1))

INVERSE HYPERBOLIC TANGENT:
     ARGTANH(X) = LOG((1+X)/(1-X))/2

INVERSE HYPERBOLIC SECANT:
     ARGSECH(X) = LOG((SQR(-X*X+1)+1)/X

INVERSE HYPERBOLIC COSECANT:
     ARGCSCH(X) = LOG(SGN(X)*SQR(X*X+1)+1)/X
```

```
INVERSE HYPERBOLIC COTANGENT:
     ARGTANH(X) = LOG((X+1)/(X-1))/2


A MOD B:
     MOD(A) = INT((A/B-INT(A/B))*B+Ø.Ø5)*SGN(A/B)
```

CHAPTER 10
SOUND GENERATION COMMANDS

In MPF-III BASIC, six commands are provided to generate sound effects. The six commands are SONG, BASS, TEMPO, INSTR, PLAY, and EFFECT. They can be used in conjunction with user-specified codes for a wide variety of sound effects.

10.1 SONG

Execution mode: imm & def
Format: SONG P1, P2

Where
P1: Specify the tone to be generated.
P2: Specify the length of the tone (beat).

The tone and the beat of a sound are represented by codes. Table 10-1 is the Tone Table and Table 10-2 the Beat Table.

		DO	RE	MI	FA	SO	LA	SI
Low Range		11	12	13	14	15	16	17
Tones	#	21	22	*	24	25	26	*

		DO	RE	MI	FA	SO	LA	SI
Middle		31	32	33	34	35	36	37
Tones	#	41	42	*	44	45	46	*

		DO	RE	MI	FA	SO	LA	SI
High Range		51	52	53	54	55	56	57
Tones	#	61	62	*	64	65	66	*

REST	Terminating code	
	BASIC	ASSEMBLY
00	100	FF

Note that the terminating code is stored at the end of a song.

Table 10-1 The Tone Table

	1/16	1/8	1/4	1/2	3/4	1	1-1/2	2	3	4
	𝅘𝅥𝅱	𝅘𝅥𝅰	𝅘𝅥𝅯	𝅘𝅥𝅮	𝅘𝅥𝅮.	𝅘𝅥	𝅘𝅥.	𝅗𝅥	𝅗𝅥.	𝅝
	70	71	72	73	74	75	76	77	78	79

Table 10-2 The Beat Table

[Example 1]

```
SONG      31,  75

          DO   One beat
```

[Example 2]

```
10        SONG  31, 75
20        SONG  32, 75
30        SONG  33, 75
40        SONG  34, 75
```

This example causes your MPF-III to play four tunes -- DO, RE, MI, FA.

[Example 3]

```
10        FOR H=1 TO 14
20        READ A,B
30        SONG A,B
40        NEXT
50        DATA 31, 75, 32, 75, 33, 75, 34, 75, 35, 75,
               36, 75, 37, 75, 51, 75, 52, 75, 53, 75,
               54, 75, 55, 75, 56, 75, 57, 75
```

This example will play 14 notes.

10.2 BASS

Excution mode: imm & def
Format: BASS P1

P1 in the above command line is used to select the rythm to be used. The value of P1 ranges from Ø through 8. BASS Ø is used to stop a rythm. Upon power-up, BASS Ø is selected by default.

Ø	1	2	3	4	5	6	7	8
Stop	Waltz	Rumba	Tango	March	Cha-cha	Blues	Rock	Swing

[Example 4]

```
 5      BASS 2
10      FOR H=1 TO 40
20      READ A,B
30      SONG A,B
40      NEXT
50      DATA 33, 75, 36, 73, 36, 75, 33, 75, 32, 75,
             31, 73, 32, 73, 31, 73, 17, 73, 16, 75,
             13, 75, 16, 75, 17, 75, 31, 75, 17, 75,
             16, 73, 17, 73, 31, 73, 32, 73, 33, 78,
             33, 75, 36, 73, 36, 73, 33, 75, 32, 75,
             31, 73, 32, 73, 31, 73, 17, 73, 16, 75,
             33, 75, 16, 75, 17, 79, 31, 75, 17, 75,
             16, 75, 16, 73, 16, 73, 16, 78,100,100,
```

Execute the above program! Note that the last two bytes of the buffer should be stored with the code "100" to terminate the song.

10.3 TEMPO

Execution mode: imm & def
Format: TEMPO P1

The above command is used to select the tempo to be used. The value of P1 should be in the range from 1 to 15. The tempo table is shown as follows:

Tempo	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Beat/Sec	640	320	240	200	160	120	110	100	90	80	70	60	58	55	52

If the song to be played is a march, the 160 beats/sec is frequently selected. In this case, you should enter the following command:

```
TEMPO 5
```

If tempo is not specified, TEMPO 6 is selected by default.

Now you are suggested to add the following command line

```
3 TEMPO 3
```

to the example program in Example 4 and run that modified program.

10.4 INSTR

Execution mode: imm & def
Format: INSTR P1

The INSTR command allows the user to select a piano. In the above command line the value of P1 is an integer in the range from 1 to 4.

P1 = 1 : represents that piano is selected.
P1 = 2 : represents that bell is selected.
P1 = 3 : represents that xylophone is selected.
P1 = 4 : represents that organ is selected.

After power-on, piano is selected by default.

Add the following command lines to the example program in Example 4 and run the modified program.

```
3  TEMPO  9
4  INSTR  4
```

10.5 PLAY

Execution mode: imm & def
Format: PLAY P1, P2

The PLAY directly assigns a memory space, whose starting address is specified by P1, as the buffer. P2 in the above command line allows the user to select the rythm (bass). SONG P1, P2 differs with PLAY P1, P2 in that PLAY assigns a buffer to store the codes for music and SONG P1, P2 is simply a BASIC command to which data is supplied by data entry commands.

[Example 5]

Store the following codes into the memory buffer whose starting address is 5000H.

```
5000H  31, 73, 32, 73, 33, 73, 31, 73, 31, 73,
       32, 73, 33, 73, 31, 73

5010H  33, 73, 34, 73, 35, 73, 33, 73, 34, 73,
       35, 73, 35, 72, 36, 72
```

```
5020H  35, 72, 34, 72, 33, 73, 31, 73, 35, 72,
       36, 72, 35, 72, 34, 72

5030H  33, 75, 31, 75, 31, 75, 15, 75, 31, 75,
       31, 75, 15, 75, 31, 75

5040H  FF, FF
```

Note that the memory buffer used to store the codes of the song should be ended with two bytes with each containing "FF".

Enter and run the following program.

```
10  A = 5 * 4096 + 0 * 256 + 0 * 16 + 0
20  PLAY  A,2
```

If "5000H = 20480" is known, then the following program will play the same song with xylophone accompanying with blues rythm.

```
10  TEMPO  3
20  INSTR  3
30  PLAY   20480, 6
```

10.6 EFFECT

Execution mode: imm & def
Format: EFFECT P1

P1 is entered to specify the special sound effects desired. The value of P1 ranges from 0 to 3.

0 : simulates whistling bomb
1 : simulates bomb explosion
2 : simulates the sound of laser gun
3 : simulates the sound of machine gun

[Example 6]

```
10  EFFECT   0
20  EFFECT   1
30  EFFECT   2
40  EFFECT   1
50  EFFECT   3: EFFECT 3
60  EFFECT   1
```

Run the above program!

Appendix A: Summary of MBASIC Commands

In this appendix, we have listed the commands discussed in Chapters 2 through 9 in alphabetical order. For more detailed descriptions of how they function, refer to the discussions of each of these commands in their individual chapters which are notated in brackets.

MBASIC COMMAND OR KEY		DESCRIPTION
(1)	ABS(-5.673)	[Chapter 9] Gives the absolute value of the parentheses. ABS(-5.673) will return 5.673.
(2)	arrow keys	[Chapter 4] Are program editing keys which are indicated by the left ← and right → arrows on your keyboard. The left arrow key is used to move the cursor to the left and will erase each character it moves over from the program line regardless of what it is. The right arrow key is used to move the cursor to the right and will enter each character it moves over as if you had typed it in from the keyboard.
(3)	ASC("CIRCLE")	[Chapter 5] Gives the ASCII code for the first character in the argument. In this command, 67 will be returned because the ASCII code for C is 67. A list of ASCII character codes is contained in Appendix K.
(4)	ATN(3)	[Chapter 9] Gives the arctangent, in radians of the argument enclosed in parentheses(in this example, 1.2490458 will be returned since it is the arctangent of 3.)

(5) BASS(2) The sound generator of MPF-III will generate a rhythm which is specified by the value following the BASS command.

(6) CALL -868 [Chapter 2] Executes a machine-language subroutine at the memory location whose address must be a decimal given by the user. CALL -868 will execute a subroutine located at the memory address 868 and will clear the current line from the cursor to the right margin.

(7) CHR$(89) [Chapter 5] Gives the ASCII character code that is associated with the value of the argument and must be in the range 0 through 255. CHR$(89) will return the letter Y.

(8) CLEAR [Chapter 4] Assigns 0 to all variables and sets all strings to null.

(9) COLOR=9 [Chapter 7] Sets the color for plotting in low-resolution graphics mode. COLOR=9 sets the color to green and GR sets the color to 0, which is the color black. Color names and their corresponding color codes are listed below.

0	BLACK	8	PURPLE
1	GREEN	9	GREEN
2	PURPLE	10	PURPLE
3	WHITE	11	WHITE
4	GREEN	12	WHITE
5	ORANGE	13	ORANGE
6	BLUE	14	BLUE
7	WHITE	15	WHITE

The SCRN command will give you the color of any given point you designate on the screen. For more details on SCRN, refer to item 78 in this appendix and the description of this command in Chapter 7.

(10) CONT

[Chapter 2] Causes program execution to CONTinue at the next instruction if it was interrupted by STOP, END, ctrl C or any other command which stopped the execution of the program. (NOTE: It will cause execution to continue at the next instruction and not the next line number.) If
- (a) a program line has been modified, added or deleted, or
- (b) an error message was received when execution was halted

then the CONT command will not be able to resume program execution.

(11) COS(3)

[Chapter 9] Gives the cosine of the argument enclosed in parentheses (in radians). COS(3) will return the cosine of 3, which is -0.9899925.

(12) CONTROL C

[Chapter 2] Stops the execution or listing of a program. CONTROL C also interrupts an INPUT instruction if it is the first character entered.

(13) CONTROL X

[Chapter 4] Instructs MPF-III to disregard the current line being typed and not to change or delete the existing line in the program with the same line number. At the end of the line to be ignored, a backslash(\) will appear when CONTROL X is given.

(14) DATA -28, BANANA, "KEY 10", 161.25

[Chapter 3] Creates and contains a list of elements which are used by READ statements. In this DATA statement, the first element is the integer -28; the second element is the literal BANANA; the third element is the string "KEY 10", the fourth element is the real number 161.25.

(15) DEF FN B(X) = 2*X+X

[Chapter 3] Lets the user define one-line functions which must be first defined by using a DEF statement before it can be used later on in the program. In this example, the function FN B(X) is first defined and may be used later in the program FN B(15) or FN B(-3*R-9). If FN B(15) is used, then 15 will be substituted for X in 2*X+X and the evaluation of the function will be as follows: 2*15+15 or 45. If FN B(-3*R-9) is used and if R is equal to 1Ø, then substituting 1Ø for R, the function becomes FN B(-3*1Ø-9) or FN B(-39). Evaluation of this function will give the result of 2*(-39)+(-39), which is -117.

(16) DEL 28,44

[Chapter 4] Deletes from the program the range of lines as specified by the user. DEL 28,44 will delete the lines 28 through 44. If only one line is to be deleted, (let's say line 58), then type DEL 58, 58 or type 58 and press the RETURN key.

(17) DIM NAME$(6Ø), SCORE(3Ø,3)

[Chapter 5] Upon execution of a DIM statement, space is automatically set aside for specified arrays with subscripts ranging from Ø through the given subscript. DIM NAME$(6Ø), SCORE(3Ø,3) will set aside 6Ø+1 or 61 strings of any length for NAME$ and SCORE will consist of 3Ø+1 rows(first dimension) by 3+1 columns(second dimension) or 124 real number elements. If an array has not been DIMensioned before it is used in the program, each dimension in the element's subscript will be allotted a maximum subscript of 1Ø. When RUN or CLEAR are executed, array elements are set to zero.

(18) DRAW 7 AT 4Ø,9Ø

[Chapter 8] Draws shape definition number 7 from a previously loaded shape table and draws the shape starting at x=4Ø, y=9Ø in high-resolution graphics. Before DRAW can be executed, however, specifications of the shape's color, rotation and scale must have already been given.

(19) EFFECT (1) — The sound generator of the MPF-III generates special sound effects according to this command. In this example, the sound of explosion is generated.

(2Ø) END — [Chapter 2] Causes the execution of a program to stop so that the user can regain control of the computer. Use of this command will not cause a message to be printed.

(21) ESC — The ESC key, when used in conjunction with the A, B, C, D, and I, J, K, M keys, is used to move the cursor to a desired location on the screen for program editing.

(22) EXP (aexpr) — Raise e to the power indicated by aexpr.

(23) FLASH — Set the video display to Flash Mode, e.g., the video display switches from normal to inverse and from inverse to normal back and forth.

(24) FOR...TO... — Allows the user to design a loop, and then execute the instructions of the loop for the number of times as specified by the loop counter.

(25) FRE(Ø) — [Chapter 4] Tells the user the amount of memory(in bytes) still available for use.

(26) GET RES$ — [Chapter 3] Receives a single character from the keyboard without displaying it on the TV screen and without making it necessary to press the RETURN key [←]. GET RES$ will store the typed character in the variable RES$.

(27) GOSUB 380 — [Chapter 6] Causes the program to continue program execution at the designated line number. Upon execution of a RETURN statement, the program branches to the next statement immediately after the most recently executed GOSUB. GOSUB 380 will cause the program to branch to line 380.

(28) GOTO 380 — [Chapter 6] Causes the program to branch to the designated line. GOTO 380 will cause the program to branch to line 380.

(29) GR — [Chapter 7] Clears the screen to black, moves the cursor into the text window, sets the COLOR to 0(black), leaves four lines of text at the bottom and sets the low-resolution GRaphics mode(40 by 40) for the TV screen.

(30) HCOLOR=7 — [Chapter 7] Sets high-resolution graphics color to be used for plotting to the color designated by HCOLOR. Color names and their corresponding values are listed below. (Colors denoted with * depends on TV.)

0 black	4 black
1 green	5 orange
2 purple	6 blue
3 white	7 white

(31) HGR [Chapter 7] Clears the screen to black, displays page 1 of memory on the screen, does NOT move the cursor into the text window, leaves 4 lines of text at the bottom and sets the high-resolution graphics mode(280 by 160) for the screen. HGR does not affect HCOLOR or the text screen memory.

(32) HGR2 [Chapter 7] Clears the screen to black, displays page 2 of memory on the screen, and sets full-screen high-resolution graphics mode(280 by 192). HGR2 does not affect the text screen memory.

(33) HIMEM:27495 [Chapter 2] Used to set the address of the highest memory location available to a MBASIC program (including variables). HIMEM: is used to protect an area of memory for high-resolution screens, machine-language routines or data. Resetting HIMEM: cannot be accomplished by changing or adding a program line, CLEAR, RUN, NEW, DEL or pressing the RESET key.

(34) HLIN 15,20 AT 40

[Chapter 7] Draws horizontal lines in low-resolution graphics mode, using the color specified by the COLOR statement executed. The origin (x=0,y=0) is at the top leftmost dot of the screen. HLIN 15,20 AT 40 will cause a line to be drawn from point (15,40) to point (20,40).

(35) HOME [Chapter 4] Clears all text in the window and moves the cursor to the uppermost left screen position in the text window.

(36)
```
HPLOT 15,20
HPLOT 25,30 TO 35,40
HPLOT TO 45,50
```

[Chapter 7] Used to plot dots and lines in high-resolution graphics mode using the value of the last HCOLOR statement executed. The origin (x=0,y=0) is at the top leftmost dot on the screen. HPLOT 15,20 plots a high-resolution dot at point (15,20). HPLOT 25,30 TO 35,40 plots a high-resolution line from point (25,30) to point (35,40). HPLOT TO 45,50 plots a line from the most recently dot plotted to the dot at point (45,50) using the color of the most recently dot plotted and not the last HCOLOR used.

(37)
```
HTAB 35
```

[Chapter 4] Causes the cursor to move horizontally (either left or right) to the designated column (1 through 40) on the screen. HTAB 35 will move the cursor to column 35.

(38)
```
IF SCORE < 60 THEN A=0:B=1:C=2
IF RES$ = "YES" THEN GOTO 250
IF P<MAX THEN 30
IF P<MAX GOTO 30
```

[Chapter 1 & 6] If the expression contained in the IF portion of the statement is true(that is, non-zero), then the instructions in the THEN portion of the same statement will be executed. However, if the IF portion of the statement is not true, then the instructions in the THEN portion will be ignored and program execution will continue with the next line number in the program. Alphabetic ranking is used to evaluate string expressions as shown in the first example above. Although the second, third, and fourth examples are worded differently (THEN GOTO, THEN, and GOTO), the effect to test this same condition is identical.

(39) INPUT C%
INPUT "TYPE NAME FOLLOWED BY A COMMA THEN SCORE"; A$,X

[Chapter 3] INPUT C% will print a question mark and wait for you to type a number, which will be assigned to the integer variable C%. The INPUT "TYPE NAME FOLLOWED BY A COMMA THEN SCORE"; A$, X command will print the optional string following the INPUT command contained in quotes and will wait for you to type a name (which will be assigned to the string variable A$) followed by a comma, and then a score (which will be assigned to the real variable X). By inserting commas or pressing the RETURN key ⏎, you can separate multiple entries to the INPUT statement.

(4Ø) INSTR(1) — The sound generator of the MPF-III selects a musical instrument according to this command. In this example, piano is selected.

(41) INT(NUM) — [Chapter 9] Gives the largest integer which is either less than or equal to the given argument. If INT(2.389) were evaluated, then 2 is returned. If INT(-45.123345) were evaluated, then -46 is returned.

(42) INVERSE — [Chapter 4] Is used to set the video mode for computer output so that black letters will be printed on a white background. To have the usual white letters displayed on a black background, use NORMAL.

(43) IN# — This command is used to select an input slot.

(44) LEFT$("MPF-III",3)

[Chapter 5] Displays the specified number of leftmost characters from the string. LEFT$("MPF-III",3) will return MPF (the three leftmost characters).

(45) LEN("HAVE A NICE DAY")

[Chapter 5] Gives the number of characters contained in a string, which must be between 0 and 255. Otherwise, you will receive an error message. LEN("HAVE A NICE DAY") will return 15.

(46) LET P = 58.012
Q$ = "EXCELLENT"

[Chapter 3] The value of the string or expression to the right of the equal sign (=) is assigned to the variable name to the left of the equal sign in a LET statement. If desired, LET can be left out of the statement. Therefore, 58.012 will be assigned to the variable P and the string EXCELLENT will be assigned to Q$.

(47) LIST
LIST 100 - 1500
LIST 100,1500

[Chapter 4] Returns all or part of the program currently in memory. Use of the first example above cause your MPF-III to list the entire program. The second example will return lines 100 through 1500 and the third example will have the same effect as the second example. If you would like to list the program from the beginning of the program through line 100, type LIST -100. If you would like to list the program from line 100 through the end of the program, type LIST 100-. LIST can be interrupted by typing CONTROL C.

(48) LOAD [Chapter 2] This command can be used to read in an MBASIC program from cassette tape and place it in memory. Although the prompt character "]" will not be displayed, MPF-III will signal the user when it has started reading in a program by the first beep and when it has finished reading in a program by the second beep. After the program has been read, the prompt character "]" will be displayed. Otherwise, an error message will appear if an error was encountered (in APPLE format).

(49) LOG(2) [Chapter 9] Gives the natural logarithm of \aexpr\. LOG(2) will cause .693147181 to be returned.

(50) LOMEM: 1135 [Chapter 2] Sets the address of the lowest memory location available to a BASIC program and protects variables in safe areas of memory set aside from high-resolution graphics with large amounts of memory.

(51) MID$ ("HAVE A NICE DAY",6)
MID$ ("HAVE A NICE DAY",6,6)
[Chapter 5] Returns a substring In the first example, it will return the string starting from the sixth character and return A NICE DAY. In the second example, it will return the string starting from the sixth character and return the six characters : A NICE.

(52) NEW [Chapter 2] Used to delete the current program and all of its variables so that a new program may be entered.

(53) NEXT [Chapter 1 & 6] Refer to the description for the FOR...TO...STEP command in chapter 6.

(54) NORMAL [Chapter 4] Used to return the screen display to the normal white letters on a black background for input and output.

(55) NOTRACE [Chapter 2] Used to turn off the TRACE mode. Refer to the TRACE command for more details.

(56) ON AB GOSUB 130,150,300,10,200

[Chapter 6] Used to run a GOSUB to the line number specified by the value of the arithmetic expression immediately following ON. In the example, if AB is 1, GOSUB 130 will be executed. If AB is 2, GOSUB 150 will be executed, etc. If the value given for AB is 0 or greater than the number of the listed line numbers in your program, program execution will continue with the next line number.

(57) ON AB GOTO 150, 300, 450, 600, 550

[Chapter 6] Is the same as the ON AB GOSUB command discussed above except that this command will execute a GOTO to the line number specified by the arithmetic expression immediately following ON.

(58) ONERR GOTO 200

[Chapter 6] Can be used in a program to prevent the program from stopping execution when an error occurs. Upon execution of this command, ONERR GOTO will set a flag for the program to perform an unconditional jump to the specified line number after GOTO (which is 200 in the example) when an error occurs.

(59)	PDL(Ø)	This command returns the current value of the specified game control (paddle). The range of the value must be from Ø through 255, while that of the game control number from Ø to 3
(6Ø)	PEEK (25)	[Chapter 2] Used to return the contents (in decimal) of the byte at the indicated decimal address given in parentheses (in this example, PEEK will return whatever is contained in the address 25).
(61)	PLAY A,B	Plays a song automatically. "A" represents the starting address and "B" the rythm of the song to be played.
(62)	PLOT 16, 29	[Chapter 7] Used in the low-resolution graphics mode to plot a dot at the indicated location (in this example, a dot will be placed at the location with x-coordinate 16 and y-coordinate 29). If a color is not specified, the color used will be black. Otherwise, the color is set by the value used by the last COLOR statement executed.
(63)	POKE -28524,Ø	[Chapter 2] Will place the binary equivalent of the second argument specified after the comma (which is Ø in this instance) into a memory location whose address is specified by the first argument (-28524).
(64)	POP	[Chapter 6] Will "pop" a RETURN address off the top of the stack of RETURN addresses. The next RETURN following a POP will cause the program to branch to one statement beyond the second most recently executed GOSUB statement.

(65) POS(Ø) [Chapter 4] Will tell the user what the present horizontal position of the cursor is (usually from Ø through 39, if Ø represents the rightmost position). Whatever is contained in the parentheses is unimportant, but it must be able to be evaluated by MBASIC.

(66) PR# Assigns the output to a slot.

(67) PRINT
PRINT C$;"A=";B
[Chapter 3] In the first example, MPF-III will perform a line feed and a return when this command is given. In the second example, the semicolons are used to separate items in the list to be printed, indicating that the items are to be printed right next to each other. If it is desired for the items to be printed in separate fields, then commas should be used to separate the items. For instance, if C$ is "CAT" and B is 9, then

CATA=9

will be printed.

(68) READ X, Y%, Z$
[Chapter 3] Used to set the value of the first variable in the first READ statement to the value of the first element contained in the DATA statement, and the second variable to the second element in the DATA statement,and so on. In this example, a number must be assigned to X, another number assigned to Y%, and a string assigned to Z$.

(69) RECALL AB

[Chapter 5] Returns an MBASIC numeric (real or integer) array AB from cassette tape. The variable name used to RECALL an array need not be the same as the variable name used to STORE the array on tape. However, the dimensions used to RECALL an array must be the same as the dimensions used to STORE an array. It is necessary first to dimension the array AB in the program with a DIM statement. In this example, MPF-III will return the array elements AB(Ø), AB(1), ...and so on. Upon execution of RECALL, you will hear a beep representing the beginning and another beep representing the end. If AB does not have a subscript, it will not be influenced. Interruption of RECALL can only be accomplished by typing RESET.

(7Ø) REM THIS IS AN MPF-III MANUAL

[Chapter 4] Allows the user to insert comments or remarks into a program.

(71) REPEAT

Repeat the character which is pressed simultaneously with the REPT key until the REPT key is released.

(72) RESTORE

[Chapter 3] Used to return the data list pointer back to the beginning of the data list so that when the READ instruction is executed again, the data starting from the beginning of the list will be read.

(73) RESUME

[Chapter 6] Can be placed at the end of an error-handling routine to resume program execution after an error has been encountered.

(74) RETURN [Chapter 6] Can be used to make the program branch to the statement right after the last GOSUB executed.

(75) RIGHT$("SCRATCHES",5)
[Chapter 5] Will return the specified number of characters in a string counting from the right. In this example, TCHES will be returned.

(76) RND(5) [Chapter 9] Will generate a random real number greater than or equal to Ø and less than 1. If RND(Ø) is used, then the last used random number will be returned. If a negative argument is used to generate a certain random number, all random numbers will be the same each time RND is used with that particular argument and, consecutive RND's using positive arguments will follow a certain, repeatable sequence. When a positive argument is used with RND, then RND will produce a new random number every time it is used except if it was used in a sequence with a negative argument.

(77) ROT=32 [Chapter 8] Will set the orientation of high-resolution shapes drawn using DRAW or XDRAW. ROT=32 will cause the shape to be drawn rotated 18Ø degrees clockwise (in other words, upside down). Using ROT=Ø will cause the shape to be drawn in its original orientation. ROT=16 will draw the shape rotated 9Ø degrees clockwise and so on. This process starts all over again at ROT=64.

(78) RUN 100 — [Chapter 2] Will clear all variables, pointers, and stacks as well as initiate execution of the program at the specified line number (which is 100 in this example). If a line number is not specified, then execution of the program will start with the lowest program line number contained in the program.

(79) SAVE — [Chapter 2] Used to save a program on cassette tape. The beginning and end of the recording will be indicated by "beeps" and when recording, the RECORD and PLAY buttons must be depressed. (Uses APPLE format.)

(80) SCALE = 20 — [Chapter 8] Sets the size of high-resolution graphics shapes to be drawn using DRAW and XDRAW. SCALE=1 means that the shape is to be reproduced exactly the same as the original dot by dot. SCALE =255 means that each plotting vector is to be extended 255 times its original size. NOTE: SCALE=0 is the maximum size a shape can be drawn and does not represent a single dot.

(81) SCRN (20,40) — [Chapter 7] Returns the color code of the indicated point (enclosed in parenthesis) in the low-resolution graphics mode. In this example, the color of the dot at the location x=20, y=40 will be given.

(82) SGN(NUM) — [Chapter 9] Returns a certain value depending on the value of the argument. If the argument is negative, -1 is returned. If the argument is 0, then 0 is returned. If the argument is positive, 1 is returned.

(83) SHLOAD — [Chapter 8] Will load a high-resolution graphics shape table from cassette tape. When the shape table is loaded, it is first loaded just below HIMEM: and then HIMEM: is used to protect the shape table when it is set to the location just below the shape table.

(84) SIN(2) — [Chapter 9] Will give you the sine of the argument in radians. In this example, .9Ø9297427 will be returned.

(85) SONG A,B — Plays the sound as specified by A and B. "A" specifies the tune and "B" the beat of the sound to be played.

(86) SPC(9) — [Chapter 4] Required to be used with the PRINT instruction to specify how many spaces are to be inserted between the most recently PRINTed item and the next item to be printed if semicolons are used before and after the SPC command.

(87) SPEED=3Ø — [Chapter 4] Used to set the speed at which characters are to be sent to your input/output device or the screen. The slowest speed is Ø and the fastest is 255.

(88) SQR(3) — [Chapter 9] Will return the positive square root of the argument enclosed in parentheses. In this example, 1.732Ø5Ø8 will be returned.

(89) STOP — [Chapter 2] Used to halt execution of an MBASIC program. Upon execution of STOP, MPF-III will tell you which line number executed the STOP and return control of the computer to you.

(90)	STORE AB	[Chapter 5] Used to save the indicated array AB on cassette tape. Upon execution of STORE, you will receive beeps at the beginning and end of the recording. In this example, AB(Ø), AB(1), AB(2)... etc. are elements which will be STOREd on tape. STORE has no effect on the variable AB. Refer to RECALL in this appendix.
(91)	STR$(35.67)	[Chapter 5] Will give you the string which stands for the value of the argument. STR$(35.67) will return 35.67.
(92)	TAB(3Ø)	[Chapter 4] Can be used to place the cursor at a specified location on the current line. Usually used with the PRINT statement and will move the cursor depending on the value of the argument which must be in the range Ø through 255. If the value of the argument is greater than the current cursor position, then the cursor is moved to the specified position. If the value of the argument is less than the current value of the cursor position, the the cursor is not moved. If the value of the argument is Ø, then the cursor will be moved to position 256.
(93)	TAN(3)	[Chapter 9] Will give the tangent of the argument in radians. TAN(3) will return -Ø.1425465.
(94)	TEMPO A	The tempo of the song to be played by the sound generator is determined by this command. The tempo is expressed in beat/minute. For example, the tempo of a march is normally 18Ø/minute.

(95) TEXT [Chapter 7] Will put the screen in the usual full-screen text mode (4Ø characters per line and 24 lines) regardless of which graphics mode the screen was being used in.

(96) TRACE [Chapter 2] Will return the line number of each statement as it is executed. RUN, CLEAR, DEL or reset cannot interrupt TRACE. To interrupt TRACE, use NOTRACE.

(97) USR(3) [Chapter 2] Will cause the program to branch to a machine-language subroutine and pass the argument (3 in this example) to the accumulator. The argument is then placed into a floating-point accumulator (locations \$9D through \$A3) after being evaluated and MPF-III will do a JSR to location \$ØA. A JMP must already be in locations \$ØA through \$ØC to the beginning of the machine-language subroutine. The floating-point accumulator will then contain the return value for the function after USR is executed. Do an RTS to return to MBASIC.

(98) VAL ("-3.7E4A5PKB")
[Chapter 5] Can be used to convert a string to a numeric value. In the example, MPF-III will try to interpret the string "-3.7E4A5PKB" up to the first non-numeric character and give you the value of that number which is -37ØØØ. If there is no number before the first non-numeric character, then Ø will be given.

(99) VLIN 40,60 AT 30

[Chapter 7] Is used to draw a vertical line on the screen in low-resolution graphics mode using the color used in the last COLOR statement executed. In this example, a vertical line will be drawn from the point (30,40) to the point (30,60).

(100) VTAB (10)

[Chapter 4] Will position the cursor at the specified line in the current column. VTAB can only move the cursor vertically from line 1 to line 24. In this example, the cursor will be placed at line 10 in the current column occupied by the cursor.

(101) WAIT 16000,255
WAIT 16000,255,0

[Chapter 2] Can be used to stop an MBASIC program until a specific condition is met by a certain memory location. In this example, 16000 is the decimal address of a memory location to be tested in order to check when particular bits are high(1) or low(0). The binary values of 255 will determine whether or not you are interested in the corresponding bit in the memory location. If the binary value of 255 is a 1, then it means you are interested, if it is a 0 then it means to skip over it. The third argument(0) will tell the computer whether you are waiting for a 1 or a 0 in the memory location. If a third argument is not specified, then it will be given a default value of 0. When the bit in the second argument matches the state of the bit given in the third argument, the WAIT is complete and program execution will resume.

(102) XDRAW 2 AT 100,150

[Chapter 8] Used to draw a high-resolution graphics shape on the screen and can erase the shape if it is used a second time with the same parameters. In this example, shape definition 2 will be taken from the previously defined shape table and will be drawn at the location x=100, y=150. Colors used to draw the shape will be the complement of the color already existing at that point.

Appendix B: Summary of Most Commonly Used Metasymbols and Their Definitions

The execution modes, formation and explanation of commands which are represented by metasymbols were previously itemized in Chapter 1. However, for your convenience and quick reference, we have placed the metasymbols and their definitions in this appendix as listed below.

imm: immediate-execution mode

def: deferred-execution mode

|: alternatives are separated by this metasymbol

{}: items enclosed in this metasymbol can be repeated

[]: items enclosed in this metasymbol are optional

\ : items whose values are to be used are enclosed in \ \. The value of X is written \X\.

name: variable name

name%: integer variable name

name$: string variable name

aop: arithmetic operator (+,-,*,/)

alop: arithmetic logical operator

aexpr: arithmetic expression

literal: literal (consisting of characters)

character: character (includes letters of the alphabet, numbers, and special symbols)

string: string (consisting of characters which are preceded and followed by quotes("))

digit: digit (numbers from 0 to 9)

sop: string operator (concatenation is represented by plus sign(+))

sexpr: string expression

expr: expression (includes arithmetic expressions and string expressions)

linenum: line of instruction

prompt character: prompt on screen as "]".

reset: a press of the RESET key on the keyboard

return: a press of the carriage return key

$\boxed{\hookleftarrow}$ on the keyboard

ctrl: holding down the CONTROL key while the following named key is typed

subscript: subscript (used to specify a particular entry in an array and is later changed into an integer when used)

Appendix C: Reserved Words in MBASIC

When referring to the BASIC language used with MPF-II, we will often refer to it as MPF-III BASIC or MBASIC. This section contains a list of the reserved words which are used in MBASIC.

&	HCOLOR=	NEXT	SAVEA
ABS	HGR	NORMAL	SAVET
AND	HGR2	NOT	SCALE=
ASC	HIMEM:	NOTRACE	SCRN(
AT	HLIN	ON	SGN
ATN	HOME	ONERR	SHLOAD
CALL	HPLOT	OR	SIN
CHR$	HTAB	PEEK	SPC(
CLEAR	HC	PLOT	SPEED=
COLOR	IF	POKE	SQR
CONT	INPUT	POP	STEP
COS	INT	POS	STOP
DATA	INVERSE	PRINT	STORE
DEFW	LEFT$	PRTON	STR$
DEL	LEN	PRTOFF	TAB(
DIM	LET	READ	TAN
DRAW	LIST	RECALL	TEXT
END	LOADA	REM	THEN
EXP	LOADT	RESTORE	TO
FN	LOG	RESUME	TRACE

FOR	LOMEM:	RETURN	USR
FRE	MID$	RIGHT$	VAL
GOSUB	MA	RND	VLIN
GOTO	MP	ROT=	VTAB
GR	NEW	RUN	WAIT
			XPLOT

In MBASIC, each of the reserved words listed above will only use up one byte of storage in memory. However, keep in mind that every character which is not used as a part of a reserved word will take up one byte of storage.

Appendix D: Screen Editor

The ESC key, when used in conjunction with several other keys, provides screen editing functions. Once the ESC key is pressed, the MPF-III will enter the Edit Mode. These edit functions enable the user to modify BASIC programs easily. These keys are listed as follows:

1. ESC : Cursor movement key
2. <- : Backspace key (left arrow key)
3. -> : Typeover key (right arrow key)
4. ↑ : Up-arrow key
5. ↓ : Down-arrow key

A. ESC

The ESC key is used to move the cursor when used in conjunction with the A, B, C, D, and I, J, K, M keys. Once the ESC key is pressed, the MPF-III will enter into the Edit Mode immediately. The key press sequence is to press the ESC first and then press one of the other keys. The functions which can be achieved by the ESC key are listed below:

ESC A : Move the cursor one position to the right
ESC B : Move the cursor one position to the left
ESC C : Move the sursor down one line
ESC D : Move the cursor up one line

Note that when one of the A, B, C, D keys is pressed following a press of the ESC key, the MPF-III will exit from the Edit Mode. The following key press sequence can be used to move the cursor consecutively:

ESC I : Move the cursor up
ESC J : Move the cursor to the left
ESC K : Move the cursor to the right
ESC M : Move the cursor down

Note that pressing the ESC or any keys other than the I, J, K, M when the MPF-III is in the Edit Mode will cause the MPF-III to exit from the Edit Mode.

B. The Typeover Key -- ->

This key can not only move the cursor to the right but also copy the characters, symbols, or programs into the memory. This function is achieved by pressing the ESC first and then the SPACE key.

C. The Backspace Key -- <-

After you typed a wrong key (but before the carriage return key is pressed), you may type this key to back space to where the wrong key is pressed and correct the error. Each time this key is pressed, the cursor moves one position to the left and the character backspaced is deleted from the input buffer for the keyboard.

D. Up-arrow Key

This key moves the cursor up. Each time this key is pressed, the cursor is moved up one line. This key can be typed continously.

E. Down-arrow Key

This key moves the cursor down. Each time this key is pressed, the cursor is moved down one line. This key can be typed continously.

Three more functions can be achieved by the ESC key. They are listed as follows:

ESC E : Delete an entire line
ESC F : Clear the data after the cursor
ESC @ : Clear the screen and home the cursor (but the program stored in the RAM is not affected.)

Appendix E: PEEK, POKE and CALL

MBASIC has a few extra features which can be used through the PEEK, POKE or CALL commands. (Note: some of them may have the same effect as other MBASIC commands.)

In MPF-III, simple switching actions are ordinarily address dependent. In other words, any command containing that address will have an identical effect on the switch. For example, if the following command was given:

```
POKE -16304,0
```

you will receive the same effect by POKEing that address using any number from 0 through 255, or by PEEKing that address using the following command:

```
X=PEEK(-16304,0)
```

This instruction is not applicable, however, to commands in which you must POKE the required address with a specific value that sets a margin or moves the cursor to a particular position.

E.1 HOW TO SET THE TEXT WINDOW

This section will tell you how to set the text window. The following POKE commands can be used to set the size of the window which displays and scrolls text on the screen.

```
100 POKE 32,L    (to set the left margin)
200 POKE 33,W    (to set the line width)
300 POKE 34,T    (to set the top margin)
400 POKE 35,B    (to set the bottom margin)
```

These commands will be discussed in further detail later.

If you set the text window, whatever is remaining on your screen will not be cleared and the cursor will not be moved into the text window (to accomplish this, use HOME, VTAB, or HTAB). When VTAB is used, the text window is totally ignored and text appearing above the window is displayed normally, however, any text below the window will be displayed on one line. Like VTAB,

HTAB is able to move the cursor out of the text window just long enough to allow one character to be printed. When setting the text window, the line width is immediately changed, however, MPF-III does not know that the left margin has been changed until the cursor tries to return to the left margin.

Any text which is shown on the TV screen is simply a replica of what is contained in a certain section of memory. The TV screen will always inspect this same section of text and look at what MPF-III has written down. You indicate to MPF-III where you want the text to be written down in memory when you change the text window. Changing the text window will work all right provided the portion of text you have specified is still in the normal text area. If the left margin is set to 255(40 is the maximum due to te fact that the screen is only 40 printing positions wide), MPF-III will assume that you want to input text beyond the normal memory area set aside for text(which is not displayed on the screen). To avoid destroying parts of your program or any information which may be contained in memory areas outside of the text memory area, abstain from setting the text window beyond the usual 40-character by 24-line screen as much as possible.

The text window can be set by using the POKE commands as shown below:

<u>COMMAND</u>	<u>ACTION</u>
POKE 32,L	Can be used to set the left margin to the value given by L, which must be between 0 and 39, inclusive(0 is the leftmost position). MPF-III will not know that you have changed the left margin of the text window until the cursor tries to return to the left margin. NOTE: This command will not change the width of the window. In other words, the right margin will be moved the same number of spaces the left margin was moved. It is recommended that you first reduce the window width then change the left margin so that nothing in your program or in MBASIC is destroyed.
POKE 33,W	Can be used to set the width (that is, the total number of characters per line) to the value given by W, which must be in

	the range of 1 through 40. NOTE: If W is set to 0, then MBASIC may be destroyed. Therefore, do not set W to 0 under any circumstances.
POKE 34,T	Can be used to set the top margin to the value given by T which must be in the range 0 through 23 (0 is the top line of the screen). NOTE: Avoid setting the top margin of the window(T) lower than the bottom margin(B).
POKE 35,B	Can be used to set the bottom margin to the value given by B, which must be in the range 0 through 24 (24 is the bottom line). NOTE: Avoid setting the bottom margin lower than the top margin(T).

E.2 OTHER COMMANDS RELATED TO TEXT, THE TEXT WINDOW AND THE KEYBOARD

This section of this appendix discusses other commands which may be used to change the conditions of text, the text window, and the keyboard.

COMMAND	ACTION
CALL -936	Will clear all the characters contained in the text window and move the cursor to the top leftmost printing position of the window. The same effect can be achieved by using HOME or TEXT.
CALL -958	Will clear all the characters contained in the text window starting from the current cursor position and ending at the bottom margin. NOTE: The position of the cursor will determine what effect this command may have on the text. If the cursor position is above the text window, then the cursor will be cleared to the right with the left and bottom margins appearing as though the top margin is above the cursor. Do not use this command if the cursor is below the bottom margin of the text window. As a result of

using this command under these circumstances, the bottom line of the text window will generally be cleared while inserting one line of text-window width at the cursor position.

CALL -871 Will clear the current line starting from the cursor position to the right margin.

CALL -922 Will issue a line feed and has the same effect as executing a CONTROL J.

CALL -912 Will move the text up one line, that is, each line contained in the specified window will be moved up one line. The previous top line will no longer exist and what was formerly the second line now becomes the first line. The bottom line becomes a blank line. Any characters not within the specified window will not be changed.

NOTE: The following program

```
10 FOR I=768 TO 781
20 READ A
30 POKE I,A
40 NEXT I
50 RETURN
60  DATA  32,67,240,170,144,3,
          32,27,240,138,141,
          16,3,96
```

will read in a code from the keyboard by using the following method:

(1) Set the subroutine program's last instruction
(2) Use CALL 768:X=PEEK(784) to read the keyboard. If is X>127, then it means that a key has been pressed and X will be increased to 128 by the value of the ASCII code of the key pressed.

E.3 COMMANDS CONCERNING CURSOR POSITION AND MOVEMENT

CH=PEEK(36) Will return the current horizontal position of the cursor on the screen and

assign it to the variable CH (which must be in between Ø through 39, inclusive). CH stands for the position of the cursor relative to the left-hand margin of the text window as set by the command POKE 32,L. For example, if the command POKE 32,3 was used to set the left margin, then the leftmost character in the window is at printing position 4 in respect to the left edge of the screen. If PEEK (36) was used to check the current position of the cursor, then the cursor is in printing position 11 from the left edge of the screen and at printing position 4 from the left margin of the text window. (Keep in mind that the leftmost position is Ø and not 1.)

POKE 36,CH — Can be used to cause the cursor to move to a position that is printing position CH+1 from the left margin of the text window. (For example: POKE 36,Ø will cause the next character from the left margin of the text window to be printed.)

CV=PEEK(37) — Will return the current vertical position of the cursor and assign the variable CV to it. The variable CV stands for the absolute vertical position of the cursor and is not looked at in respect to the top or bottom margins of the text window. Therefore, CV=Ø is considered to be the top margin and CV=23 is considered to be the bottom margin.

POKE 37,CV — Can be used to move the cursor to the absolute vertical position as designated by the variable CV. As discussed above, CV=Ø is the top margin and CV=23 is the bottom margin.

E.4 COMMANDS USED WITH GRAPHICS

In order to display text and graphics on the TV screen, MPF-III's memory has been designed to contain two areas for DISPLAY positions as listed below:

(1) 2000 - 3FFF
(2) A000 - BFFF

In MPF-III, the commands TEXT, GR, HGR, HGR2 must use DISPLAY positions in memory. Because of this, there are two commands which can be used to set these DISPLAY positions as follows:

(1) MA Used to set the display buffer to 2000-3FFF
(2) MP Used to set the display buffer to A000-BFFF

NOTE: In MBASIC, commands will operate in certain modes as described below:

1. TEXT operates completely in the TEXT mode
2. GR operates partially in low-resolution graphics mode (first 20 lines) and in the TEXT mode (last 4 lines)
3. HGR operates partially in high-resolution graphics mode (first 20 lines) and in the TEXT mode (last 4 lines)
4. HGR2 operates partially in high-resolution graphics mode (first 23 lines) and in the TEXT mode (last line)

REMEMBER: (a) In HGR, to display numbers and symbols in the first 20 lines, you must first set VTAB and HTAB to print out what you want printed.

For example, when the following program

```
]10 HGR
]20 VTAB 12:HTAB5:PRINT "ABC"
```

is executed, then ABC will be printed on the screen in line 12 at position 5.

As another example, when the program

```
]10 HGR
]20 PRINT "ABC"
```

is executed, then ABC will be printed on the screen in line 12 at position 1.

(b) GR, HGR, HGR2 and HOME can only clear the window area, and cannot clear the entire screen.

E.5 COMMANDS ASSOCIATED WITH GAME CONTROLS AND SPEAKER

COMMAND	ACTION
X=PEEK(-16336)	Shakes up the speaker once to induce a click from the speaker.

E.6 COMMANDS CONCERNING ERRORS

COMMAND	ACTION
X=PEEK(218)+PEEK(219)*256	Assigns X to the line number of the statement where an error was encountered upon the execution of an ONERR GOTO statement.
IF PEEK(216)] 217 THEN GOTO 2000	Denotes that an ONERR GOTO statement has been executed when bit 7 at memory location 222(ERRFLG) has been set true.
POKE 216,0	Used to clear the value contained in ERRFLG so that the user will receive the normal error messages.
Y=PEEK(222)	Assigns a code to variable Y in order to indicate what type of error caused a jump to ONERR GOTO. Types of errors and their respective Y values are as listed below:

Y VALUE	TYPE OF ERROR
0	NEXT without FOR
16	Syntax
22	RETURN without GOSUB
42	Out of DATA
53	Illegal Quantity

69	Overflow
77	Out of Memory
90	Undefined Statement
107	Bad Subscript
120	Redimensioned Array
133	Division by Zero
163	Type Mismatch
176	String Too Long
191	Formula Too Complex
224	Undefined Function
254	Bad Response to an INPUT Statement
255	CONTROL C Interrupt Attempted

```
POKE 768,104 : POKE 769,168 : POKE 770,104 :
POKE 771,166 : POKE 772,223 : POKE 773,154 :
POKE 774,72  : POKE 775,152 : POKE 776,72  :
POKE 777,96
```

Sets up a machine-language subroutine at location 768 so that it can be used in an error-handling routine. Can also be used to clear up some ONERR GOTO problems which may occur with the PRINT statement ?OUT OF MEMORY ERROR messages and must be used with CALL 768 in the error-handling routine.

Appendix F: Storing the Results (or Data) of an Executed Basic Program on Cassette Tape-(Using the STORE and RECALL Commands)

Most ordinary users can store BASIC programs on cassette tape after writing up a program. However, many users do not know how to store the result or data of an executed program on tape. Since storing the result or data of a program on tape makes it readily accessible to the user for the next time it must be used, let us now look at how to accomplish this task through the use of the STORE and RECALL commands.

(1) STORING AND RECALLING DATA ON CASSETTE TAPE

Suppose we want to store the test scores of a history class on tape. Assuming that there are only 10 students in the history class, let us first execute the following sample program to input the scores of the ten students into an array:

```
]LIST

10 DIM A(10)
20 FOR I=1 TO 10
30 PRINT "STUDENT'S SCORE";I;
40 INPUT A(I)
50 NEXT

]RUN

STUDENT'S SCORE 1?10
STUDENT'S SCORE 2?20
STUDENT'S SCORE 3?30
STUDENT'S SCORE 4?40
STUDENT'S SCORE 5?50
STUDENT'S SCORE 6?60
STUDENT'S SCORE 7?70
STUDENT'S SCORE 8?80
STUDENT'S SCORE 9?90
STUDENT'S SCORE10?100
```

Now that the students' records have been placed in the array dimensioned by DIM A(10), let us now look at how to store this data on tape.

First of all, make sure that your tape recorder's electrical cord is properly plugged in (one end should be plugged in the MPF-III mainboard and the other end should be plugged in your tape recorder's MIC jack), then press the RECORD button and type STORE A after the cursor is displayed on your screen. Once this is done, your data will already have been stored on tape.

To return the data we have just stored on tape, we can use the RECALL command. When executing the RECALL command, the data will be returned from the tape upon execution of the following sample program:

```
]LIST 100,

100 DIM A(10)
110 PRINT "PLEASE PRESS RECORDER'S 'PLAY'"
120 RECALL A
130 FOR I=1 TO 10
140 PRINT I, A(I)
150 NEXT

]RUN 100
PLEASE PRESS RECORDER'S PLAY
```

After one data element has been RECALLed, MPF-III will continue to repeatedly execute the program until all the data elements have been read.

Now press your tape recorder's PLAY button (one end of your tape recorder's cord should be plugged in MPF-III's mainboard and one end should be plugged in your tape recorder's EAR jack). After all of the data have been read, then the following will be printed:

```
1       10
2       20
3       30
4       40
5       50
6       60
7       70
8       80
9       90
10      100
```

At this time, the students' records may be RECALLed again.

(2) STORING AND RECALLING STRINGS

In addition to storing and recalling numbers, we also often store and recall strings. Now let's suppose we would like to store the names of the 10 students on tape.

Because STORE and RECALL fundamentally can only store numbers, we must first convert each character of the string to a number before storing it on tape. (Suppose each student's name cannot exceed 9 characters.)

First of all, let's input the students' names by executing the following program:

```
]LIST

10 DIM A(10,10)
20 FOR I=1 TO 10
30 INPUT "STUDENT'S NAME:   ";A$
40 FOR J=1 TO LEN(A$)
50 K$=MID$(A$,J,1):K=ASC(K$)
60 A(I,J)=K
70 NEXT
80 A(I,J+1)=0:NEXT

]RUN
STUDENT'S NAME:   NAME1
STUDENT'S NAME:   NAME2
STUDENT'S NAME:   NAME3
STUDENT'S NAME:   NAME4
STUDENT'S NAME:   NAME5
STUDENT'S NAME:   NAME6
STUDENT'S NAME:   NAME7
STUDENT'S NAME:   NAME8
STUDENT'S NAME:   NAME9
STUDENT'S NAME:   NAME10
```

After all the names have been inputted, this data has now been placed in DIM A and at the same time by typing STORE A on your MPF-III keyboard, pressing the RECORD button on your tape recorder, you can store the data on tape.

Explanation of the above program:

LINE 1Ø: is used to dimension an array A with (1Ø,1Ø) to store the names

LINES 2Ø-3Ø: are used to input the names of each student

LINES 4Ø,5Ø,6Ø: are used to convert the names to numbers to be stored in the array specified by the DIM statement.

LINE 8Ø: is used to set the end of name input, which is represented by Ø.

To recall the names of the students, we can again use the RECALL command. When RECALL is executed, the data will be returned from tape upon execution of the following sample program:

```
]LIST 1ØØ

1ØØ DIM A(1Ø,1Ø)
11Ø DIM A$(1Ø)
12Ø PRINT "PLEASE KEYIN PLAY KEY"
13Ø RECALL A
14Ø FOR I=1 TO 1Ø
15Ø FOR J=1 TO 1Ø
16Ø K=A(I,J)
17Ø IF K=Ø THEN J=1Ø:GOTO 19Ø
18Ø A$=A$+CHR$(K)
19Ø NEXT
2ØØ A$(I)=A$
21Ø A$="":NEXT
22Ø FOR I=1 TO 1Ø:PRINT I,A$(I):NEXT

]RUN
PLEASE KEYIN PLAY KEY
```

Execution of this program will be repeatedly continued until all the data elements have been read.

Now press the PLAY button on your tape recorder and the data will then be printed on your screen as follows:

1	NAME1
2	NAME2
3	NAME3
4	NAME4
5	NAME5
6	NAME6
7	NAME7
8	NAME8
9	NAME9
10	NAME10

At this time, the names of the 10 students can be RECALLed again.

Explanation of the above program:

LINE 100: is used to set an array dimensioned by DIM A(10,10) to store the ASCII codes for the student's names

LINE 110: is used to set an array dimensioned by DIM A$(10) to store the students' names

LINES 120-130: is used to recall data from the tape

LINES 140-190: is used to convert the ASCII codes to characters.

LINE 220: is used to print out the names

The two sample programs above have been simplified for the purposes of illustrating how to use STORE and RECALL. Hence, if you have a lot of data and if it is fairly complex, please refer to Section 5.8 of this manual.

Appendix G: Reading Error Messages

As it is commonly known among programmers, errors often occur in computer programs and must be corrected before the program can be properly executed. If you recall, we mentioned that interruption of program execution can be avoided by using an error-handling routine like the ones in Chapter 6. However, although the ONERR GOTO statement may "catch" most errors in your program (depending on how good your error-handling routine is), you may discover that MPF-III may instead exit your program, return an error message and display the prompt character on your screen. This process of locating and correcting errors in a program is known as the process of "debugging" your program. To help you to debug your programs, MPF-III has special error messages which will help you to locate your errors. Therefore, this appendix contains an alphabetized list of some common error messages and their definitions frequently encountered by programmers.

The general format that MBASIC uses to print out error messages depends on which execution mode was used to execute the instruction. In the immediate-execution mode, an error message will appear as shown below:

?(name of error message) **ERROR**

In the deferred-execution mode, an error message will appear with the error message as well as the line number so that you would be able to locate the error. Error messages of errors which occurred in the deferred-execution mode will have the following format:

?(name of error message) **ERROR IN** (line number in which error occurred)

Listed below are some of the most frequently encountered messages and their definitions:

ERROR MESSAGE	DEFINITION
BAD SUBSCRIPT	Attempt to reference an array element which exceeds the dimensions of the array. For example, when dimensions of an array are DIM A(2,2). Using

LET A(1,1,1)=2 will cause an error.

CAN'T CONTINUE	Occurs when an attempt is made to execute a program which is: (1) a program in which an error has occurred and an attempt to resume execution was made, (2) a program to which a line of instruction has been deleted or added, (3) a program which does not exist.
DIVISION BY ZERO	Any number divided by zero(Ø) is undefined and is considered to be an error.
FORMULA TOO COMPLEX	Occurs upon execution of three or more IF...THEN statements having the same format.
ILLEGAL DIRECT	Using INPUT, DEF FN or DATA statement is not allowed in an immediate-execution statement.
ILLEGAL QUANTITY	An argument to be used in a math or string function was not in the specified range. this type of error may be encountered as a result of: (a) an illegal argument was used with LEFT$, MID$, ON...GOTO, PEEK, POKE, RIGHT$, SPC, TAB, WAIT or any of the graphics functions. (b) a negative array subscript (c) a negative or zero argument is used with LOG (d) a negative argument is used with SQR (e) A^B where A is a negative number and B is a non-integer
NEXT WITHOUT FOR	Means that the variable used in a NEXT statement and its corresponding FOR statement do

	not match or FOR was used without NEXT; and vice-versa.
OUT OF DATA	Attempt was made to execute a READ statement when all of the DATA statements in the program have already been used. Also can mean that too much data was read in by the READ statement or the data in the DATA statements were insufficient.
OUT OF MEMORY	This error may occur due to the following: (1) FOR loops were more than 10 levels deep (2) GOSUB's were more than 24 levels deep (3) expressions used are too complex (4) parentheses were more than 36 levels deep (5) LOMEM: was set lower than the current value (6) LOMEM: was set too low (7) LOMEM: was set too high (8) variables in the program are too numerous to be contained in memory (9) a program which is too large is to be stored in memory
OVERFLOW	MBASIC's number format could not contain the final result of an arithmetical calculation because it may have been too large. If it should happen that the number is too small, then no error message is given and MPF-III will print a zero(0).
REDIM'D ARRAY	Another dimension statement having the same dimensions as another array used in the same program was encountered.
RETURN WITHOUT GOSUB	A RETURN statement was executed, but MPF-III could not find its GOSUB statement

STRING TOO LONG	A string in your program is more than 255 characters long
SYNTAX ERROR	Is given when the syntax rules used in MBASIC are not followed. Examples of this are: inserting a illegal character in an instructon line or forgetting a parenthesis in an expression.
TYPE MISMATCH	Occurs when the variable name given in an assignment statement does not match with the argument given. For instance, this error will occur when a numeric variable name was given a string argument.
UNDEF'D STATEMENT	The line number specified in a GOTO, GOSUB or THEN does not exist in the program.
UNDEF'D FUNCTION	An attempt was made to reference a non-existent function which must have been defined by the user.

Appendix H: Cutting Down on Your Program Execution Time

By speeding up your program, you can cut down considerably on execution time. Listed below are some hints to help you improve your MBASIC program execution time. If you have read Appendix I, you will notice that some of these hints are the same ones which may be used to save space. This simply indicates that acceleration of the speed of your programs and improvement of the efficiency of memory used by your programs can be accomplished simultaneously.

(1) By placing frequently-referenced lines as early as possible in the program, you can improve your program execution time. When MBASIC encounters a statement such as "GOTO 1000" during program execution, it will search the entire program for the referenced line (in this case, line 1000) starting from the line with the lowest number. So it would be advantageous to place frequently-referenced lines near the beginning of your program rather than at the end.

(2) Another hint for decreasing program execution time is to use NEXT statements without the index variable. when NEXT I is executed, the computer checks to see if the variable specified in NEXT is the same variable specified in the FOR statement; therefore, NEXT I takes more time to execute than NEXT.

(3) The MOST SIGNIFICANT SPEED HINT BY A FACTOR OF 10 is probably to use variables instead of constants. Because converting a constant to its floating point (real number) representation is more time-consuming, you can save a lot of time by using variables since it takes less time to fetch the value of a simple or array variable. This technique will especially help to decrease execution time in programs which contain FOR...NEXT loops or other codes that are executed repeatedly.

(4) During an execution of a BASIC program, variables which are encountered at the beginning are allotted at the start of the variable table. For instance, if the following statement is the first statement executed:

```
10 X=A:Y=B:Z=0
```

the variable X will be placed first, Y placed second, and Z placed third in the variable table. References made later on in the program to variable X will cause MBASIC to look for only one entry in the variable table to find X, two entries to search for Y and three entries to find Z, and so on.

Appendix I: Saving Memory Space

If you do not have a large system and would like to use less memory space to store your programs, the hints listed below may be of some help to you. Nevertheless, it is recommended that you use the first two space savers only when you are confronted with serious space limitations.

HELPFUL HINTS TO SAVE MEMORY SPACE
****

(1) One way to save space is to USE SEVERAL STATEMENTS IN EACH LINE. Each program line has an overhead of 5 bytes, of which 2 bytes are used to store the line number of the line in binary. Since your line numbers are stored in binary form, it will still take two bytes to store your line number regardless of how many digits you have in your line number. Therefore, if you put as many statements as possible into each line (which may consist up to 239 characters), you can cut down on the number of bytes used in your program. Although this technique seems to be a good way to save space, it not only makes editing and making changes to your program very difficult but also makes the program very hard to read and understand.

(2) Another way to save space is to DELETE ALL REM STATEMENTS, because each REM statement takes up space in memory. Keep in mind; however, that deleting all your REM statements makes your program harder to read and understand.

(3) By USING INTEGER INSTEAD OF REAL ARRAYS wherever possible, you can also save space in memory. (For more details, refer to the STORAGE ALLOCATION INFORMATION section in this appendix.)

(4) A fourth way to save sace is to USE VARIABLES INSTEAD OF CONSTANTS. For instance, if the constant 3.14159 is used 10 times in your program, you can save space and speed up program execution by using the variable P to represent the constant 3.14159. It takes more bytes to store 3.14159 ten times than just storing the variable P.

(5) If an END statement at the end of a program is not required, DELETION OF AN END STATEMENT is another way to save space in memory.

(6) Another space-saver can be utilized by RE-USING THE SAME VARIABLES, because this can decrease the

number of variables used in a program.

(7) The USE OF GOSUB's for the purpose of executing sections of program statements whih perform the same tasks will also save memory space.

(8) Another method is to USE THE ZERO ELEMENTS OF MATRICES; for example, A(Ø), B(Ø,X).

(9) In the reassignment of string values, you can save memory space by using a statement in the form

X = FRE(Ø)

which will intermittently delete old strings from memory. For example, if the string A$ = "BLUE" is reassigned to A$ = "PINK", the old string "BLUE" is not deleted from memory. However, with the use of a statement such as X = FRE(Ø), you will not need to be concerned about old strings not being erased from memory.

STORAGE ALLOCATION INFORMATION
****

When you store your program in memory, a certain amount of space must be allocated to different expressions contained in your program. Storage allocation is expressed in bytes and will be discussed in this section.

Simple (non-array) variables and array variables consist of three types: (1) real variables, (2) integer variables, and (3) string variables. The amount of storage space taken up by smple variabes are usually 7 bytes and storage space used by array variables may vary. The storage allocation for simple (non-array) variables and array variables have been summarized in the table below.

TYPE OF VARIABLE		STORAGE ALLOCATION (in bytes)
SIMPLE (NON-ARRAY) VARIABLES (7 bytes)	REAL VARIABLES	2 bytes for VARIABLE NAME 5 bytes for VALUE (1 for exponent, 4 for mantissa)
	INTEGER VARIABLES	2 bytes for VARIABLE NAME 2 bytes for VALUE 3 bytes consisting of 0's
	STRING VARIABLES	2 bytes for VARIABLE NAME 1 byte for length of the string 2 bytes for a pointer to location of string in memory 2 bytes consisting of 0's
ARRAY VARIABLES	REAL VARIABLES (12 bytes)	2 bytes for VARIABLE NAME 2 bytes for SIZE OF ARRAY 1 byte for NUMBER OF DIMENSIONS 2 bytes for SIZE OF EACH DIMENSION 5 bytes for EACH ARRAY ELEMENT
	INTEGER VARIABLES	2 bytes for EACH ARRAY ELEMENT
	STRING VARIABLES	3 bytes for EACH ARRAY ELEMENT (1 for length, 2 for pointer)

String variables (regardless if it is a simple or an array variable) use one byte of memory for each character in the string. The location of the strings are determined by the order of its occurrence in the program, starting at HIMEM:.

When a new function is set by a DEF statement, 6 bytes are used to store the pointer to the definition.

FOR, GOTO and other such reserved words as well as COS, INT and other such names of intrinsic functions use only one byte of program storage. All other characters contained in the program take up one byte of program storage each.

When you initiate execution of a program, space is dynamically allocated on the stack as described below:

(1) Every active FOR ... NEXT loop takes up 16 bytes

(2) Every active GOSUB (which has not RETURNed yet) takes up 6 bytes.

(3) Every time the computer comes across a parenthesis in an expression, it uses 4 bytes to store the parenthesis and uses 12 bytes to store the temporary result calculated in an expression.

Appendix J: ONERR GOTO Error Codes and Assembly Language Fix

ONERR GOTO ERROR CODES
****

CODE	ERROR DESCRIPTION
Ø	NEXT without FOR
16	syntax
22	RETURN without GOSUB
42	Out of DATA
53	Illegal Quantity
69	Overflow
77	Out of Memory
9Ø	Undefined Statement
1Ø7	Bad Subscript
12Ø	Redimensioned Array
133	Division by Zero
163	Type Mismatch
176	String Too Long
191	Formula Too Complex
224	Undefined Function
254	Bad Response to Input Statement
255	CONTROL C Interrupt Attempted

ONERR GOTO ASSEMBLY LANGUAGE FIX
****

MACHINE-LANGUAGE		65Ø2 ASSEMBLY LANGUAGE	
Hexadecimal (used while in Monitor)	Decimal (used while in MBASIC)	Instruction	Comments
68	1Ø4	PLA	Instruction to place top byte of stack in Accumulator
A8	168	TAY	Instruction to save it in Y index register
68	1Ø4	PLA	Instruction to place next byte of stock in Accumulator
A6	166	LDX$DF	Instruction to set pointer for ONERR
DF	223		
9A	154	TXS	as stack address
48	72	PHA	Instruction to push the contents of the saved stack onto "ONERR" stack
98	152	TYA	
48	72	PHA	
6Ø	96	RTS	Instruction to return to MBASIC

Appendix K: ASCII Character Codes

DEC = ASCII decimal code
HEX = ASCII hexidecimal code
CHAR = ASCII character name
n/a = not accessible directly from the MPF-III keyboard

DEC	HEX	CHAR	WHAT TO TYPE
0	00	NULL	n/a
1	01	SOH	ctrl A
2	02	STX	ctrl B
3	03	ETX	ctrl C
4	04	ET	ctrl D
5	05	ENQ	ctrl E
6	06	ACK	ctrl F
7	07	BEL	ctrl G
8	08	BS	ctrl H or <-
9	09	HT	ctrl I
10	0A	LF	ctrl J
11	0B	VT	ctrl K
12	0C	FF	ctrl L
13	0D	CR	ctrl M or RETURN
14	0E	SO	ctrl N
15	0F	SI	ctrl O
16	10	DLE	ctrl P
17	11	DC1	ctrl Q
18	12	DC2	ctrl R
19	13	DC3	ctrl S
20	14	DC4	ctrl T
21	15	NAK	ctrl U or ->
22	16	SYN	ctrl V
23	17	ETB	ctrl W
24	18	CAN	ctrl X
25	19	EM	ctrl Y
26	1A	SUB	ctrl Z
27	1B	ESCAPE	n/a
28	1C	FS	n/a
29	1D	GS	n/a
30	1E	RS	n/a
31	1F	US	n/a
32	20	SPACE	SPACE
33	21	!	!
34	22	"	"
35	23	#	#
36	24	$	$
37	25	%	%
38	26	&	&

39	27	'	'
4Ø	28	(	(
41	29	)	)
42	2A	*	*
43	2B	+	+
44	2C	,	,
45	2D	-	-
46	2E	.	.
47	2F	/	/
48	3Ø	Ø	Ø
49	31	1	1
5Ø	32	2	2
51	33	3	3
52	34	4	4
53	35	5	5
54	36	6	6
55	37	7	7
56	38	8	8
57	39	9	9
58	3A	:	:
59	3B	;	;
6Ø	3C	<	<
61	3D	=	=
62	3E	>	>
63	3F	?	?
64	4Ø	@	@
65	41	A	A
66	42	B	B
67	43	C	C
68	44	D	D
69	45	E	E
7Ø	46	F	F
71	47	G	G
72	48	H	H
73	49	I	I
74	4A	J	J
75	4B	K	K
76	4C	L	L
77	4D	M	M
78	4E	N	N
79	4F	O	O
8Ø	5Ø	P	P
81	51	Q	Q
82	52	R	R
83	53	S	S
84	54	T	T
85	55	U	U
86	56	V	V
87	57	W	W

88	58	X	X
89	59	Y	Y
9Ø	5A	Z	Z
91	5B	[	n/a
92	5C	\	n/a
93	5D	]	n/a
94	5E	^	^
95	5F	_	n/a
96	60		
97	61		
98	62		

Appendix L: Decimal Tokens for Keywords

DECIMAL TOKEN	KEYWORD
128	END
129	FOR
130	NEXT
131	DATA
132	INPUT
133	DEL
134	DIM
135	READ
136	GR
137	TEXT
140	CALL
141	PLOT
142	HLIN
143	VLIN
144	HGR2
145	HGR
146	HCOLOR=
147	HPLOT
148	DRAW
149	XDRAW
150	HTAB
151	HOME
152	ROT=
153	SCALE=
154	SHLOAD
155	TRACE
156	NOTRACE
157	NORMAL
158	INVERSE
160	COLOR=
161	POP
162	VTAB
163	HIMEM:
164	LOMEM:
165	ONERR
166	RESUME
167	RECALL
168	STORE
169	SPEED=
170	LET
171	GOTO
172	RUN
173	IF
174	RESTORE
175	&
176	GOSUB
177	RETURN
178	REM
179	STOP
180	ON
181	WAIT
182	LOAD(A)(T)
183	SAVE(A)(T)
184	DEF
185	POKE
186	PRINT
187	CONT
188	LIST
189	CLEAR
190	GET
191	NEW
192	TAB(
193	TO
194	FN
195	SPC(
196	THEN
197	AT
198	NOT
199	STEP
200	+
201	-
202	*
203	/
204	^
205	AND
206	OR
207	>
208	=
209	<
210	SGN
211	INT
212	ABS
213	USR
214	FRE
215	SCRN(
217	POS
218	SQR
219	RND
220	LOG
221	EXP
222	COS
223	SIN
224	TAN
225	ATN
226	PEEK
227	LEN
228	STR$
229	VAL
230	ASC
231	CHR$
232	LEFT$
233	RIGHT$
234	MID$
235	HC
236	PRTON
237	PRTOFF
238	MP
239	MA

Appendix: M: MBASIC Zero Page Usage

LOCATION (in hex)	USE
\$Ø → \$5	Location containing jump instructions to continue in MBASIC.
\$A - \$C	Location used to store USR function's jump instruction.
\$D - \$17	Location to store general purpose counters/ flags for MBASIC.
\$2Ø - \$4F	Reserved locations for MPF-III system monitor.
\$5Ø - \$61	MBASIC's general pupose pointers.
\$62 - \$66	Location containing the result of the last multiplication or division operation.
\$67 - \$68	Location for pointer to start of program.
\$69 - \$6A	Location for pointer to beginning of simple variable space and to end of program plus 1 or 2 locations unless changed with the LOMEM: statement.
\$6B - \$6C	Location for pointer to beginning of array space.
\$6D - \$6E	Location for pointer to end of numeric storage in use.
\$6F - \$7Ø	Location for pointer to start of string storage. From this location, strings are stored beginning at this point to the end of memory.
\$71 - \$72	Location for a general pointer.
\$73 - \$74	Highest location in memory available to MBASIC plus one. This location is set to the highest RAM memory location available upon initial entry to MBASIC.
\$75 → \$76	Location for the currrent line number of line being RUN.
\$77 - \$78	Location for "old line number" and is set up by a CONTROL C, STOP or END statement. The line number at which execution was interrupted will be given by this locatic
\$79 - \$7A	Location for "old text pointer" which points to the memory location containing the nex statement to be executed.
\$7B - \$7C	Location containing the current line number from which DATA is being READ.
\$7D - \$7E	Location which points to absolute location in memory from which DATA is being READ.
\$7F - \$8Ø	Location for the pointer to current source of INPUT. During an INPUT statement, this location is set to \$2Ø1 and is set to the DATA in the progam it is READing from during a READ statement.
\$81 - \$82	Location which holds the most recently used variable's name.
\$83 - \$84	Location containing the pointer to the most recently used variable's value.
\$85 - \$9C	Location for general usage.

\$9D - \$A3	Location containing the main floating point accumulator.
\$A4	Location for general use in floating point math routines.
\$A5 - \$AB	Location for secondary floating point accumulator.
\$AC - \$AE	Location for general usage flags/pointers.
\$AF - \$BØ	Location containing the pointer to end of program which cannot be changed by LOMEM:
\$B1 - \$C8	Location containing CHRGT routine which is a subroutine called by MBASIC everytime it needs another character.
\$B8 - \$B9	Location for the pointer to last character obtained through the CHRGT routine.
\$C9 - \$CD	Location for random number.
\$DØ - \$D5	Location for high-resolution graphics scratch pointers.
\$D8 - \$DF	Location for ONERR pointers/scratch.
\$EØ - \$E2	Location for high-resolution graphics X and Y coordinates.
\$E4	Location for high-resolution graphics color byte.
\$E5 - \$E7	Location for general use for high-resolution graphics.
\$E8 - \$E9	Location containing pointer to beginning of shape table.
\$EA	Location for collision counter for high-resolution graphics.
\$FØ - \$F3	Location for general use flags.
\$F4 - \$F8	Location for ONERR pointers.

Appendix N: Memory Map

<u>Memory Range</u>

```
$ 0   -- $ FF  RAM (Zero Page)
$100  -- $1FF  RAM (Stack Area)
$200  -- $2FF  RAM (Input Buffer for the Keyboard)
$300  -- $3FF  RAM (Memory Used by the Peripherals)
$400  -- $7FF  RAM (Memory Used by the Monitor Program)
$800  -- $1FFF RAM (Memory for Storing BASIC program)
$2000 -- $3FFF RAM (Area  for  Text,  Low-Resolution,  Hi-
                   resolution Graphics)
$4000 -- $9FFF RAM (Memory for Storing BASIC program)
$C000 -- $CFFF Hardware I/O Addresses
$D000 -- $F7FF ROM (BASIC Interpreter)
$F800 -- $FFFF ROM (Monitor Program)
```

Appendix O: Converting BASIC Programs to MBASIC

Due to the various implementations of the BASIC language by different computer companies, you will discover that some BASIC programs may first need to be converted into MBASIC before it can be RUN on your MPF-III. Take a look at the following incompatibilities you should watch out for:

(1) ARRAY SUBSCRIPTS. Your MPF-III uses parentheses with array subscripts. Replace any brackets ("[" and "]") to indicate array subscripts. MPF-III uses parentheses("(" and ")").

(2) STRINGS. When using MPF-III, you should delete all dimension statements which specify the length of strings and convert DIM M$(N,O) used in BASIC to DIM M$(O), which is the one used in MBASIC.

To express string concatenation, MPF-III uses the plus sign, "+". Therefore, if any "," or "$" are used, replace it with "+". In addition to this, MPF-III uses MID$, LEFT$, and RIGHT$ to take substrings of strings. Do not use M$(N) to access the Nth character of the string M instead use M$(N,O) to take a substring of M$ from character position N to character position O.

However, MPF-III uses MID$(M$,N,1) and MID$(M$,N,O-N+1). Some other BASIC programs use M$(N)=X$ to denote that the Nth character of string M is represented by the string X, but MPF-III uses M$=LEFT$(M$,N-1)+X$+MID$(M$,N+1). Likewise, to convert M$(N,O)=X$ so that you can use it on MPF-III, it is transformed to M$=LEFT$(M$,N-1)+X$+MID$(M$,O+1). As a result, expressions are converted as follows:

OLD (BASIC)	NEW (MBASIC)
M$(N)	MID$(M$,N,1)
M$(N,O)	MID$(M$,N,O-N+1)
M$(N)=X$	M$=LEFT$(M$,N-1)+X$ +MID$(M$,N+1)
M$(N,O)=X$	M$=LEFT$(M$,N-1)+X$ +MID$(M$,O+1)

(3) If an assignment statement such as

```
500 LET B=C=0
```

appears in a BASIC program then change it to:

```
500 C=0:B=C
```

to execute it in MPF-III BASIC.

(4) In MPF-III BASIC, the colon ":" must be used as a delimiter when several statements are used in one line. Other BASIC programs may use a slash "/".

(5) MAT functions cannot be used in MBASIC, therefore, if you encounter a MAT function in a BASIC program, you must write the entire mathematical expression out.

Appendix P: MPF-III Operators

The operators are listed vertically according to their priority from the highest to the lowest. Operators listed in the same line have the same priority. Operators of the same priority, when present at the same expression, are evaluated from left to right. Parentheses have the highest priority and or the logical OR operation has the lowest priority.

Symbol	Function
^	Exponent
-	Unary or Subtract operator
+	Unary or Add operator
*	Multiply operator
/	Divide operator
=	Assign a value
NOT	Perform a logical NOT operation
AND	Perform a logical AND operation
OR	Perform a logical OR operation
XOR	Perform a logical XOR operation
=	The equal sign
<	Less than (Relational operator)
>	Greater than (Relational operator)
<=	Less than or equal to
>=	Greater than or equal to
<>	Not equal to

Special Characters

RETURN: Every program line should be ended with a carriage return.

: : BASIC commands or instructions in the same program line should be separated by the ":" mark.

? : This mark is the equivalent to the PRINT command.

$: A string variable should be ended with a "$" sign.

% : An integer variable should be ended with a "%" sign.

MULTITECH INDUSTRIAL CORP.

Office: 315 Fu Hsin N. Rd., Taipei, Taiwan, R.O.C.
Factory: 1 Industrial E. Rd., III Hsinchu Science-based Industrial Park, Hsinchu, Taiwan, R.O.C.

DOC. NO.: M3003-8309A

MPF-III BASIC Programming Manual

Multitech